# Fast Synchronization for Shared-Memory Multiprocessors

*Philip Bitar*

December 1985

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 85.11

RIACS

Research Institute for Advanced Computer Science

# Fast Synchronization for Shared-Memory Multiprocessors

## Synchronizing Caches
## Busy-Wait Locking, Waiting, Unlocking
## Sleep-Wait and Service-Request Queuing
## Paradigm for High-Contention Atomic Operations

*Philip Bitar*
Research Institute for Advanced Computer Science
NASA Ames Research Center
*and*
Computer Science Division
University of California, Berkeley

December 19, 1985

### Synchronizing Caches
### Busy-Wait Locking, Waiting, Unlocking

The issues in *broadcast synchronization schemes for caches* are analyzed, and new methods for busy-wait locking, waiting, and unlocking are introduced. The *lock/unlock* scheme allows busy-wait locking and unlocking to occur in zero time, eliminating the need for test-and-set; while the *wait* scheme eliminates all unsuccessful retries from the switch, and allows a process to work while busy-waiting. These methods for busy-wait locking, waiting, and unlocking also integrate *processor* atomic read-modify-write instructions and *programmer/compiler* implementations of atomic, busy-wait-synchronized operations under the same mechanism, and improve the performance of both approaches to atomic operations. The evolution of broadcast schemes is also analyzed in detail.

### Sleep-Wait and Service-Request Queuing
### Paradigm for High-Contention Atomic Operations

Fast queuing operations on priority queues, including the sleep-wait operations P and V, can be executed by VLSI hardware, whose structure, function, and management are presented. This introduces a paradigm for *VLSI implementation of high-contention atomic read-modify-write operations.* The paradigm will virtually eliminate switch traffic in the execution of such operations, as well as speed up the operations themselves tremendously.

## Abstract

### Synchronizing Caches
### Busy-Wait Locking, Waiting, Unlocking

A cache may serve several purposes. In a shared-memory system in which caches serve as local memories for writable data, read/write sharing of data among the caches must be synchronized. This is an instance of the problem of *read/write synchronization of replicated data*, which entails two tasks: serializing conflicting access requests, and providing the latest version of the requested data. A cache synchronization scheme for *broadcast* systems is presented, introducing efficient methods for busy-wait locking, waiting, and unlocking. The *lock/unlock* scheme allows busy-wait locking and unlocking to occur in zero time, eliminating the need to fetch busy-wait lock bits from memory independently of the data in order to test-and-set them. The *wait* scheme eliminates all unsuccessful retries from the switch, and allows a process to work while busy-waiting. This method for busy-wait locking, waiting, and unlocking also integrates *processor* atomic read-modify-write instructions and *programmer/compiler* implementations of atomic, busy-wait-synchronized operations under the same mechanism, improving the performance of both approaches to atomic operations. Many options are possible for a cache synchronization scheme for a *broadcast* system, and will be selected according to the constraints of cost, performance, and off-the-shelf parts that will be used. The options introduced in the literature are compared and contrasted in a presentation of the *evolution* of broadcast synchronization schemes.

### Sleep-Wait and Service-Request Queuing
### Paradigm for High-Contention Atomic Operations

Fast queuing operations on priority queues, including the sleep-wait operations P and V, can be executed by VLSI hardware, thereby eliminating the need to busy wait for access to high-contention process queues and service-request queues. The structure, function, and management of the hardware queues are presented, along with their interface to the processors and the switch. The queuing hardware is relatively simple, and is suitable for VLSI design, due to the replication of cells and the simple control. In addition, this introduces a paradigm for *VLSI implementation of high-contention atomic read-modify-write operations*. The paradigm will virtually eliminate switch traffic in the execution of such operations, as well as speed up the operations themselves tremendously.

# Contents

# Detailed Contents

# A. Perspective

## A.1. Motivation

**What is the most important issue in computer architecture today?**

(I invite the reader to pause a moment and offer an answer.)

One of the most important issues is how to design multiprocessor systems that can realize the speedup potential of many processors. The major problem is how to arrange for the processors to *coordinate, or synchronize,* their activities efficiently, so that the overhead of coordination is not greater than the benefit realized by the greater concurrency. To be specific, suppose that a problem can be arranged to run faster on a multiprocessor than on a uniprocessor, but when the execution time necessary to make these arrangements is counted, the total execution time is not appreciably less than the execution time on a uniprocessor. In this case, the expense of the multiprocessor does not pay off.

**Shared-Memory Architecture.** Project Aquarius at Berkeley, lead by Professors Alvin Despain and Yale Patt, is interested in building *high-performance* systems. We believe that performance and cost are best served in a multiprocessor system by taking a shared-memory, or *processor-to-memory,* architecture to its limit, including as many processors as possible, before moving at a higher level into a message-passing, or *processor-to-processor,* architecture (Gajski, Peir 1985). Performance is better served because memory accesses in a processor-to-memory architecture tend to be more tightly timed (with respect to mean and variance) than memory accesses in a processor-to-processor architecture, though I do not believe that these correlations are logically necessary.

At the other extreme, a shared-memory, single-bus architecture offers a *low-cost* approach to building a multiple-microprocessor system, so this kind of architecture is rapidly gaining in importance (Bell et al. 1985).

In view of these two distinct interests, it is useful to identify architectural features that improve the coordination, or synchronization, of processors in a shared-memory architecture. A designer of either kind of system, then — high performance or low cost — will be able to select the features that best serve their interest.

**Prolog.** Project Aquarius is investigating the design of a high-performance, multiprocessor system to execute logic programs, currently Prolog (Dobry, Despain, Patt 1985). We have built a uniprocessor system to execute Prolog and are now designing a multiprocessor system. In order to realize the concurrency potential in logic programs, we are planning to implement predicates (procedures) as lightweight processes — all executing in the same virtual address space for a given program (Kepecs 1985). Furthermore, predicates will be producing and consuming many variable values, and they will need to synchronize this activity.

In short, we are planning for many *medium-grained, lightweight processes.* We expect a

typical program to have a large number of these processes, and the processes will be generating much synchronization activity, including frequent sleep and wakeup operations. Rumor has it that synchronization activity may occur every 10 or 20 memory references. So we are devoting a great deal of effort toward making synchronization fast. Hence this report.

## A.2. Method

**Architectural Support for Operating Systems.** Professor John Ousterhout addresses the topic of architectural support for operating systems this way.

> My advice to computer architects who want to help operating system designers is this: Don't do us any favors!

(CS252 guest lecture, U.C. Berkeley, October 14, 1985) Ousterhout is responding to the history of computer system architecture, which is cluttered with examples of complex, slow, inflexible features implemented for operating system 'support.' Many of these features were even advocated by operating system (OS) designers themselves, unable to foresee the full, negative implications the features would eventually have on the development, use, and performance of the system.

Ousterhout states that in an environment where security is not of prime concern, the only real help that computer architects can offer OS designers is to make critical, high-frequency operations *fast*. Furthermore, these hardware operations should be relatively *primitive* operations — for use by OS designers to implement higher-level operations and policies of their choosing.

To illustrate, Ousterhout favors reference bits on main memory pages (omitted from the VAX), hardware management of cache synchronization, primitive synchronization operations such as test-and-set, and fast data transfer for I/O and memory-to-memory transfer. On the other side, measurements of standard programming environments, such as the Mesa environment at Xerox PARC, indicate that process switching is relatively infrequent in these environments, while procedure calls are very frequent. So the speed of the latter, not the former, should be improved by hardware targeted for that environment.

In short, the two goals for architectural support for OSs (in a non-high-security environment) are these.

- *Performance:* A hardware feature is justified only if it speeds up a high-frequency operation.
- *Primitives:* A hardware feature should be relatively primitive, so that it can be used in a variety of ways by the OS designers to create higher-level operations and policies of their choosing.

In this report I introduce three such *primitives for performance*, and I propose an implementation for a primitive suggested by Ousterhout.

- *Cache lock state* — for broadcast switch
  - Allows zero-time busy-wait locking and unlocking, eliminating the need for fetching and test-and-setting a lock bit independently of the data it protects
  - Integrates processor atomic read-modify-write instructions and programmer/compiler implementations of atomic, busy-wait-synchronized operations under the same mechanism
- *Cache busy-wait register* — for broadcast switch
  - Eliminates all unsuccessful retries from the switch
  - Allows a process to work while busy waiting
- *Hardware queue (VLSI circuit)* — for any switch
  - Virtually eliminates switch traffic in the execution of such operations, as well as speeding up the operations themselves tremendously
  - Can be used for both sleep-wait (P and V) and service-request queues (e.g., a floating point or I/O processor)
  - Can be used for both FIFO and priority queues
  - Introduces a paradigm for implementation of any high-contention atomic read-modify-write operation
- *High-speed memory transfer operation*
  - Transfer from one memory unit to another at the clock rate, where a memory unit is a main memory unit, a cache, or an I/O-processor buffer

The first three primitives not only improve the performance of high-frequency, synchronization operations, but also integrate otherwise disparate operations under the same mechanism, thereby reducing the cost of the design and the hardware. The fourth, data-transfer, primitive was suggested by Ousterhout as greatly needed by OS designers, so I offer two possible implementations of it, one for maximal speed and one for lower cost and easier use.

**Technology.** A watchword among computer scientists is to track the trends of technology and take advantage of what technology is or will be offering. To be specific, one of the key conclusions today is that memory is relatively inexpensive, so it should be used liberally in designing computer systems.

My own vision, in this vein, is that MOS VLSI design technology is available, and is getting better fast, so hardware circuits such as sleep/priority queues — which would be unreasonably expensive if designed using standard TTL or ECL chips — are now feasible and can be strategically employed to implement high-contention atomic read-modify-write operations.

**Snapshot.** This report offers a snapshot of rapidly developing ideas, but I have attempted to stop the action and capture a clear picture for the reader. Further, the

major points in nearly every section are presented as bullets in that section. Consequently, the key features of the snapshot can be identified by paging through, aiming for the bullets.

## A.3. Overview

I currently identify three major, low-level synchronization issues for shared-memory architecture.

- Synchronization of caches
- Implementation of busy wait
- Implementation of sleep wait

**Synchronization of Caches.** Smith (1984) characterizes the issue this way:

> The solution of the multicache consistency problem for large numbers of processors is one of the *most important* current problems in computer architecture, and it is one of the major barriers to effective multiprocessing.

Specifically, processes in a shared-memory system communicate by taking *sole access* to some shared data object and writing it, leaving information in memory for another process to read. One example, typical of Prolog and dataflow, is the *producer/consumer* relationship. In this case, one process produces a value, say a variable binding, for another process, and that process, in turn, reads the value and uses it. The second process may also report back to the first process, in which case it also writes a shared-variable. Another example is the management of *service-request queues*, where one process leaves a service request for another process in the latter's request queue. The latter eventually reads the request and services it. This will typically occur among processes running on different processors. For example, a process running on a program interpreter may send a service request to a floating-point processor or an I/O processor.

In this context, the processor caches must correctly implement the read/write sharing of data that is requested by the software. Briefly, *cache synchronization* consists of this:

- Read/write sharing of replicated data among the caches

**Busy Wait, Sleep Wait.** If the wait for sole access to a shared object is expected to be short, the process will *busy wait*. That is, it will continue to *run while waiting*, though as we will see, this does not mean that it must continually test a bit while waiting. On the other hand, if the wait is expected to be long, the process will be switched out of the processor and will *sleep wait* on a process queue, allowing another process to run on that processor. However, if the hardware in a multiprocessor system does not itself implement P/V (or equivalent) operations, then by default the software must implement sleep wait using busy wait. In this case, a queue-manager process, instead of invoking hardware-queue facilities, will busy wait for access to a software-implemented process queue; and when it gains access, it will enqueue or dequeue a process, as appropriate. If semaphores are used, they will be part of the queue descriptor. In brief,

sleep wait as a *high-level* concept must be implemented using busy wait at a *lower level* if the hardware does not implement P/V operations. In this case, sleep wait does not actually avoid busy wait. Rather the hope is to reduce the time spent executing wait operations by busy waiting for access to a sleep-wait queue rather than for access to the target data.

This identifies the two reasons for using busy wait.

- A situation where busy wait is *less costly* than sleep wait
- A system where busy wait is *necessary* in order to implement sleep wait

Keep in mind that in either case, the atomic operation may be implemented by a single processor instruction or by a section of code, though the second case presents an extremely large bite for a single instruction, and will probably be implemented by several software routines.

**High-level Issues.** High-level synchronization issues address the content of the synchronization operations themselves (Gajski, Peir 1985). I will not discuss these issues, except to cite the sharing due to a producer/consumer relationship and due to service-request queues, as mentioned.

# B. Synchronizing Caches
# Busy-Wait Locking, Waiting, Unlocking

## B.1. General Concepts in Cache Synchronization



This section will develop the general concepts underlying cache synchronization. We will first look at the purpose of caches, then move on to the concept of read/write synchronization and consider the hardware role in the synchronization of caches. Next we will consider the role of broadcast in cache synchronization and the way a cache may handle irrelevant requests that come via the switch. Finally, we will look at the implications that the write policy for shared data — write-back *vs.* write-through to other caches — has on the placement of atoms in memory blocks.

### *B.1.1. Purpose of Caches*

A cache may serve several purposes in a computer system, including these:

- *High-speed memory*
- *Local memory* — in a multiprocessor system
- *Busy-wait locking/waiting* — in a broadcast multiprocessor system
- *High-speed memory-to-memory transfer*
- *Switch mediation* — especially in a broadcast multiprocessor system

**High-Speed Memory.** In a system with just one processor accessing memory — a single CPU and no I/O processors — the cache is the high-speed component of the memory hierarchy external to the CPU (Figure 1). Wilkes (1965) was one of the earliest to discuss this concept (Censier, Feautrier 1978; Denning 1970). The concept, which he called 'slave memory,' carried the notion that its management would be simple enough to be implemented in hardware and would thus be fast.

**Local Memory.** In a shared-memory system with several processors that access memory — CPUs, I/O processors, etc. — a cache is connected to a processor through a shared switch or through a private path (Figures 2,3). If connected through a shared switch, the cache serves only as the high-speed component of the memory hierarchy. But if connected through a private path, the cache also serves as *local memory* for the processor, reducing the processor's need to compete with other processors in accessing the switch and main memory. That is, if a block of data is fetched into the cache and subsequently read several times, only the first read requires fetching from main memory (or another cache) through the switch. An early implementation of this function was in the two page-table caches under the MULTICS operating system. Consistency of the caches was maintained by invalidating both caches when the page table was written by

either of the two processors (P. Denning, personal comm. 1985).

A critical issue here is the policy for updating other caches and main memory with the contents of a dirty block. An update scheme can be thought of as a variation or combination of the following two basic policies.

- *Update-by-word (write-through):* A cache updates main memory and other caches that have the block (if there are any) with a word whenever its processor writes to the cache. *On a write:*
  - Update main memory
  - Update other caches having the block (if any)
- *Update-by-block (write-back):* A cache updates another cache only when the latter requests the block. Memory may be updated at the same time, or else only when the cache must purge the block due to its own activity. When a processor writes to the cache, copies of the block in other caches are *invalidated*, rather than updated. *On a write:*
  - Do not update main memory
  - Invalidate other caches having the block (if any)

Another typical update scheme derives from uniprocessor systems and non-broadcast multiprocessor systems: update main memory (as in write-through under a uniprocessor system), but invalidate other caches (as in a non-broadcast system, where updating other caches cannot be done simultaneously).

I have introduced 'update-by-word' and 'update-by-block' as alternate names to 'write-through' and 'write-back' in order to draw attention to what I feel is the key difference between the two policies, namely, the *granularity* of the update. We will see the space and time implications of this granularity.

Specifically, *update-by-word*, or write-through, serves the function of local memory better than update-by-block if a relatively *small* number of writes to a block are made before it must be purged or be read by another cache. Whereas *update-by-block*, or write-back, serves local memory better if a relatively *large* number of writes to a block are made before it is purged or read by another cache. If a block has $n$ words, the tradeoff point will be expressed as some fraction of $n$ writes.

To illustrate, if exactly $n$ writes will be made to a block before it is purged or read by another cache, it is better to transfer the block as a whole, rather than in single-word chunks. This is because the switch traffic due to switch arbitration (if not overlapped) and due to transfer initiation, as well as the delay to those processors whose caches are updated, will be $n$ times smaller (on average) under update-by-block, while the actual bus-transfer time will be the same. A stochastic model can be devised, allowing estimation of which update policy will tend to be better for a particular system.[1]

[1] Technical note concerning *update-by-block:* A block of shared data is probably first fetched for read privilege by a cache on a processor read, so if it is present in another cache at that time, a subsequent ac-

Update-by-block for blocks that are *not shared* among caches appears to offer substantial reduction in bus traffic and improvement in performance over update-by-word, according to Norton and Abraham (1982) and Smith (1982). For blocks that are *shared* among caches, Archibald and Baer (1985) provide evidence of the opposite — that update-by-word is better than update-by-block. However, their model of how sharing is done will not occur under a properly managed update-by-block policy, as will be shown in Section B.1.4.

**Busy-Wait Locking/Waiting.** A new function introduced in this report is to have the processor data caches in a broadcast system implement busy-wait locking and waiting. Under this scheme, in order to lock a data object, a processor does not execute test-and-set on a data bit. Instead, the first block of the object is fetched with write privilege into the cache (if the block is not already locked by another cache), and its state is set to locked (Figure 4). If another cache requests the block, the cache in which it is locked reports that fact, and records that another processor is waiting (Figure 5). The requester cache, then, stores the block address with busy-wait status, and its processor waits for notification from the cache when the block has been successfully fetched and locked.

When the processor holding the lock is done with the privileged operation, it unlocks the locked block and, if there was another processor waiting, broadcasts the unlocking to all caches (Figure 6). At the next bus arbitration, all of the caches holding the block address in busy-wait status arbitrate for the bus. The winner fetches the block with write privilege, locks it, and notifies its processor. The other waiting caches continue waiting, but do not access the bus with unsuccessful retries, since they lost the critical arbitration. The details of locking and waiting will be discussed in Section B.2.2.

**High-Speed Memory-to-Memory Transfer.** Another new function introduced in this report is to have a processor cache implement high-speed transfer from one location in memory to another. As mentioned in Section A.2, this is motivated by Ousterhout's statement that operating system designers greatly need a fast memory-transfer operation. A cache can be used to implement this by reading a block from one location of memory into its assembly register and then immediately storing the block into another location of memory (Figure 7). The cache simply needs to understand a command from the processor to do this, and then, in a multiprocessor system, follow its synchronization protocol in reading and writing the block. The intelligence required of the cache is not great. Further details are given in Appendix 1.

**Switch Mediation.** The two preceding purposes point to an interesting trend that is just beginning to be explored.

---

cess will be required for each block to claim write privilege (invalidate the other copies) at the time that the block is written. Consequently, the update-by-block policy will usually incur *two* switch accesses, not just one, per block of shared data. A stochastic model for evaluating the two update policies will include all such relevant details.

- *Switch mediation:* Give to the cache (instead of to other processor hardware) functions that entail switch interactions in cases where more efficient use of time, space, or hardware will result.

This strategy is especially useful under a *broadcast system*, where the cache has hardware devoted just to monitoring the bus (or buses) and for responding to requests that are broadcast there. The example of efficient busy-wait locking and waiting was depicted above.

Another very similar example is that of *priority preemption*, discussed in Section C.1.2. In this case, each cache (or other processor hardware) monitors the bus for the priority of any process that is placed on a ready queue. If the priority of a currently running process is lower than the latter priority, the monitoring hardware arbitrates for the bus, as in efficient busy-wait, and if it wins — in this case, due to lowest priority — it interrupts its processor. The process running there is switched out, and a process is loaded from the appropriate ready queue.

### *B.1.2. Read/Write Synchronization*

**Logical Aspects.** As mentioned, cache synchronization consists of coordinating the read/write sharing of replicated data among the caches. *Read/write sharing of replicated data* entails three logical aspects.

- *Atomicity:* sole access for writers
- *Concurrency:* shared access for readers
- *Replication:* getting the latest version of the data upon access

A writer is given sole access so that its action appears atomic, all-or-none, to any subsequent reader or writer. (A writer may also read.) Sole access is necessary for primitive objects, while ordered access, analogous to pipelining, is possible for higher-order objects. Readers are given shared access in order to maximize concurrency in accessing the data. Finally, upon gaining access privilege to the data, the latest version must be fetched, wherever it may be — in any cache or in main memory.

**Implementation Requirements.** These three logical aspects of read/write synchronization reduce to two implementation requirements.

- *Serialize conflicting access requests:* write-read and write-write conflicts
- *Provide the latest version of the data:* wherever it may be

The first requirement must be met in order for the second to have meaning.

*Atomicity, Atoms.* The concept of atomicity (realized through sole access to primitive objects for a writer) is central to the problem of read/write synchronization, and can be subdivided into two distinct types: that implemented by hardware and that implemented by software. Specifically, in order for software to implement an atomic operation (insuring sole access for a writer), the hardware must provide some primitive atomic

operation for the software to use. This can be as simple as writing a single bit, as in Peterson's algorithm (Peterson, Silberschatz 1985, p. 332). But for the sake of speed, the hardware will probably provide at least a test-and-set operation or atomic swap, allowing the software to insure sole access without so many bit reads and writes as otherwise needed. In this context it becomes convenient to define two types of atomic (writable, shared) data objects.

- *Hard atom:* data object is atomized (access conflicts are serialized) by the hardware
- *Soft atom:* data object is atomized by the software

*Hardware Role.* It falls to the hardware, then, to serialize conflicting access requests only for hard atoms. This situation occurs when two different processes on two different processors simultaneously attempt to access the same hard atom (Figure 8). Providing the latest version of the data, on the other hand, is required not only for writable, shared data (hard and soft atoms), but also for writable, unshared data when a process runs on one processor and then goes to sleep and is subsequently awakened on another processor (Figure 9). In both cases, the latest version of the data must be gotten when the process accesses the data.

The *synchronization of caches,* then, reduces to the following two requirements, each having the occasions shown.

- *Serialize conflicting access requests:* hard atoms only
  - *Two different processes* on two different processors access the same hard atom.
- *Provide the latest version of the data:* all writable objects
  - *Two different processes* on two different processors access the same writable, shared data (hard or soft atom).
  - *One process* on two different processors accesses the same writable, shared or unshared, data.

### *B.1.3. Broadcast*[2]

**Full Broadcast.** Broadcast is a hardware tool that is useful in the high-speed synchronization or coordination of a group of devices, such as caches or processors. It is most simply implemented with *a bus for each broadcaster,* carrying the broadcaster's message to all devices. Hence it requires *n buses* where the capability of *n* simultaneous broadcasts is desired. Furthermore, in its full sense broadcast requires *full associativity.*

- *Full broadcast — full associativity:* A request is broadcast to all devices, since it is not known which devices may be able to service the request. Every device evaluates if it can service the request and responds appropriately.

[2] The innovative idea motivating the next section was to implement a full-broadcast cache scheme in a parallel switch, and was conceived by Professor Al Despain. George Adams and Steve Melvin offered me further perspective on the potential of the average case — as opposed to the worst case.

This stands in contrast to *partial broadcast*, in which a proper subset of devices may be addressed — only those that can service the request — so as not to interfere with the remaining devices. Unless otherwise qualified, *broadcast* will henceforth refer to *full broadcast* throughout this report.

For $n$ simultaneous broadcasts, every device considers all requests at the time that a set of broadcasts is initiated, and decides which, if any, it will service. With regard to caches in particular, every cache that has control of one of the $n$ buses at broadcast time will broadcast its request to all other caches; and every cache will consider all of the requests (except its own).

More specifically, in accessing all caches simultaneously, broadcast provides high-speed implementation of the two hardware tasks in cache synchronization.

- *Serialize conflicting access requests — hard atoms only:* Broadcast simultaneously accesses all caches that have a copy of the hard atom.
- *Provide the latest version of requested data — all objects:* Broadcast accesses the cache having the latest version of the block.

At every switch setting, each cache that has access to a memory module broadcasts its request (read/write/invalidate/etc. concerning a block address in that module) to all other caches. So for a switch allowing $n$ requesters, there must be $n$ buses, one for each requester's broadcast. Every cache will consider all $n$ requests (except its own) and will decide which, if any, it will service.

In the *worst case*, all of the requests, e.g., $n$ block fetches, will resolve to the same cache (Figure 10). The complexity of the cache will allow it to service no more than a few simultaneous requests, say one or two, however, so the other requesters will have to try again later. This is similar to the worst case for accessing main memory itself: in the worst case for main memory (ignoring the caches), all requests go to the same module. The switch will allow no more than one or two requests to the same module, however, so the other requesters will have to try again later. In short, the success of the parallel switch in accessing main memory, as well as the success of $n$-fold broadcast in accessing the caches, depends entirely on the *average case*. The parallelism should be implemented only if stochastic and simulation models indicate that the average cases of interest can realize the concurrency offered by the parallel switch and memory.

Also note that under $n$-fold broadcast, the degree of associativity for each block frame (represented in the switch directory) would be $n$, allowing $n$ simultaneous addresses to be compared. ( $n$ = min( max # requesters, # processors - 1 ) ) The scheme allowing maximal concurrency, considered by Rudolph and Segall (1984) and by Despain (personal comm. 1984), would give each processor $n$ smaller caches, one for each memory module. A weaker alternative would be $n$ cache directories, one for each memory module, where hits are arbitrated. But the latter is functionally equivalent to having a cache directory with $n$-way associativity for each block frame, as just described.

The principles of broadcast will be illustrated throughout this report by a system with a

single bus rather than a system with a parallel switch. The reasons are that broadcast in a single-bus system is more feasible in terms of cost; it is easier to understand, and to illustrate; and it has been implemented in single-bus systems.

**Irrelevant Requests.** The disadvantage of *full broadcast* is that a device, in particular a cache, will be bombarded with irrelevant requests that arrive via the switch. A cache cannot service such a request when the relevant block does not have the appropriate status there. This bombardment can interfere with a processor's use of its cache. Consequently, under full broadcast, a cache is given two control units, one for processor registers and one for switch requests; it is also given a separate directory for handling requests from the switch (Goodman 1983), or else the cache directory is given dual-ported read capability (Borriello et al. 1985).

Another approach to the problem of irrelevant requests is that of *partial broadcast,* in which a list of all relevant caches is kept in main memory for each memory block (Censier, Feautrier 1978). The list is a bit vector indicating the caches in which the block resides. So when a request is broadcast by a memory module, this presence list also sent to all caches simultaneously, on the data lines of the switch, and each cache determines if the request is relevant simply by checking to see if its bit is set. However, disadvantages of this scheme are that memory is more complicated, both in terms of block structure and control, and performance is substantially lower since all inter-cache transactions must be mediated by main memory. For example, if a cache simply purges a clean block, it must still access main memory so that its bit in the block's presence list is cleared. In addition, this scheme does not lend itself to extension of more caches, since the block in main memory and the switch datapath must be enlarged, if they are too small. (The latter is probably large enough, though, namely four or eight bytes.)

### *B.1.4. Placement of Atoms in Blocks*

Another issue in systems in which data caches serve as local memories is the placement of atoms in memory blocks, which are the unit of cache management. The unit of main-memory management may be variable-size logical units (segments) or fixed-size physical units (pages) or a combination (Denning 1970). The same decision presents itself in cache management, although the speed-cost tradeoff is viewed with heavier bias toward *speed,* following Wilkes' original concept, and has resulted in strictly fixed-size units (blocks). Fixed-size units are simpler, and hence faster, to place, and it is simpler to locate specific cells within them. And since spatial proximity or locality generally holds, block-based designs are more useful than word-based designs.

The placement of atoms in blocks, then, emerges as a critical concern. The appropriate strategy depends on the update policy — update-by-block or update-by-word.

- *Under update-by-block (write-back):* An atom should begin at a block boundary, and any memory block on which it resides should be devoted to it; unfortunately, internal fragmentation will result.
- *Under update-by-word (write-through):* Atoms can be placed together in blocks, so should be placed to improve locality of reference, if possible, as well reduce internal fragmentation.

Just the same, in both cases it is desirable to fit an atom entirely on one block, if possible, so that a block fetch will not occur while the lock is locked and thereby prolong the locking time. This will tend to generate internal fragmentation if block size is large relative to atom size.

More specifically, under an *update-by-block policy,* when a cache obtains sole access to an atom on a memory block, it simultaneously obtains sole access to access the rest of the data on the block. Consequently, if another cache desires access to the rest of the data while the first cache desires access to the atom, the two access requests will conflict and will be serialized, needlessly reducing concurrency. In the worst case, the two respective processors will work on two different atoms in the same block, and the block will be thrashed from cache to cache, as each cache, in turn, fetches the block for sole access (Figure 11). It follows, then, that under an update-by-block policy, an atom should begin at a block boundary, and any memory block on which it resides should be devoted entirely to it.[3] Archibald and Baer (1985) show through simulation that an update-by-word policy indeed performs better than an update-by-block policy when the latter causes a block to be thrashed from cache to cache.

However, this placement policy will tend to cause *internal fragmentation* of blocks, worse if block size is large relative to atom size. The effect of fragmentation on cache *space* is to waste part of a block, thereby increasing contention for cache space, which will be worse for smaller caches (given a particular block size). In this age of switch bottlenecks and relatively inexpensive memory, however, the worst effect will be on *performance,* since an entire block must be fetched in order to obtain an atom on it, thereby increasing switch traffic and, if the processor must wait for the entire block to arrive, increasing the processor wait.

The performance cost of internal fragmentation can be reduced by fetching only part of the block, namely a fixed-size sub-block *transfer unit.* Each transfer unit for a block would have a valid bit and a dirty bit, allowing independent fetching and flushing, respectively. Studies have shown that the concept of sub-block transfer unit improves performance in the access of unshared data (Smith 1982; Goodman 1983; Hill, Smith

[3] This is true unless one can show that for specific atoms of interest, such contention would be small, and that the complexity of implementing the exception is worth the cost. Appendix 4 also shows that under a non-broadcast scheme, the blocks of a soft atom must be devoted entirely to the atom to insure correctness — otherwise flushing one atom can overwrite the other in main memory.

1984) Hill and Smith offer extensive data showing the tradeoff between bus traffic and miss rate: as the size of the transfer unit increases, approaching the block size, the bus traffic increases but the miss rate decreases. Note, by the way, that the worst internal fragmentation will occur for single-bit atoms, in particular, test-and-set bits. However, the new locking method introduced in this report eliminates the need for test-and-set bits, thereby eliminating the space and performance cost of this method of busy-wait locking and unlocking.

**Update-by-Block (Write-Back)** *vs.* **Update-by-Word (Write-Through).** The issue of update-by-block *vs.* update-by-word now acquires new connotations: the issue should be evaluated for unshared and shared data separately, and for the transfer unit size rather than the full block size (if the two are different). The evidence cited in Section B.1.1 shows that update-by-block significantly reduces bus traffic for unshared data, which implies that the number of writes to a block tends to be significantly larger than the tradeoff point. This advantage will not be so great for shared data, however, if atoms tend to be small relative to block size. But with transfer units smaller than block size, it may be possible for update-by-block to regain an advantage over update-by-word.

In short, it is necessary to determine the size that hard and soft atoms will tend to be (hard atoms will tend to be smaller than soft atoms) and to determine their relative frequency and how many writes each will tend to get. Based on this, the appropriate transfer-unit size for shared data can be determined and compared to the size desired for unshared data. If the two are close enough, then update-by-block will be appropriate for both shared and unshared data. Otherwise, update-by-word will be appropriate for shared data, as in the DEC Firefly and the Xerox Dragon (reported in Archibald, Baer 1985).

## B.2. Broadcast Systems for Cache Synchronization



As mentioned in Section B.1.3, broadcast will be considered in the context of a single-bus system, for simplicity. First we will look at the necessary components of a broadcast protocol, then we will turn to the states and mechanics of the new version introduced here. Next we will consider efficient busy-wait locking and unlocking and will analyze problems with implementing the lock state, and will provide solutions. We will then look at efficient busy wait, a new method using a busy-wait register, as well as other methods. Finally we will retrace the evolution of broadcast protocols implemented under an update-by-block (write-back) policy, considering the schemes of Goodman (1983), Frank (1984), Papamarcos and Patel (1984), Katz et al. (1985), and the present.

### *B.2.1. Protocol Components*

A broadcast cache-synchronization protocol must include the following components.

- Requests for a processor to give to its cache
- Requests for a bus master (cache or processor) to broadcast on the bus
- Responses of a cache to the processor and bus requests
- Responses of main memory to the bus requests
- States and state transitions for a cache block

A cache synchronization protocol also includes three basic request-response sequences, along with states and state transitions for a requested block.

**Request-Response Sequences.** The request-response sequences are the following.

- *Processor request to cache, no bus access needed:* A processor makes a request to its cache, and the cache services the request without needing to access the bus.
- *Processor request to cache, bus access needed:* A processor makes a request to its cache. The cache needs to access the bus in order to service the request, so it arbitrates for the bus, and when master, broadcasts its request on the bus. All other caches and main memory respond appropriately, one of which provides the latest version of the data, if requested. The cache then replies to its processor.
- *I/O-processor request to bus:* An I/O processor bypasses its cache (if it has one) and broadcasts an I/O request on the bus. All caches and main memory respond appropriately, one of which provides the data, if requested.

Keep in mind that in order to reduce traffic on the bus, it will be assumed that the cache policy for updating main memory will be update-by-block (write-back) rather than

update-by-word (write-through), as discussed in Sections B.1.1 and B.1.4. Just the same, as pointed out in Section B.1.4, several parameters of a system must be estimated to determine if update-by-block will be as effective for shared data as it is for unshared data. Also, in order to reduce contention for the caches generated by irrelevant requests under the full broadcast scheme, it will be assumed that each cache has dual directories or a dual-ported-read directory, as discussed in Section B.1.3.

**States.** We will consider the following eight states for a block in a cache, and Section B.2.3 will show how the states in other protocols relate to these eight.

Invalid

Read
Read, Source, Clean
Read, Source, Dirty

Write, Source, Clean
Write, Source, Dirty

Lock, Source, Dirty
Lock, Source, Dirty, Waiter

The following is a key to the word meanings, which will be clearly illustrated by examples in Section B.2.2.

| | |
|---|---|
| *Invalid* | Meaningless |
| *Read* | Read-only privilege (*shared-access* privilege — for multiple readers) |
| *Write* | Read and write privilege (*sole-access* privilege) |
| *Lock* | Read and write privilege, locked by the cache |
| *Source* | Source of latest version of block<br>• Location of clean/dirty status for the block<br>• When the block is fetched by another cache, the source provides it and its current clean/dirty/lock status<br>• When purging the block, the source flushes it if dirty |
| *Dirty/Clean* | Block was/not written by some processor since read from memory into the cache system |
| *Waiter* | Another processor is waiting for the block to be unlocked |

Keep in mind that the goal is to enumerate promising possibilities. Although eight states are of interest here, ***not necessarily all states will be implemented in a particular system.***

First, if the system is built from *off-the-shelf parts,* the capabilities of those parts may limit the implementation. The most critical component, here, is probably the *bus:* the capability of the bus can greatly limit the designer of a cache synchronization protocol, because it may not allow them to signal all of the codes that they would like to. The most notable example is probably that of Goodman (1983). Goodman designed his cache

protocol for the original Multibus, and this bus does not give the user a way to explicitly signal block invalidation while fetching a block. As a result, Goodman used the write operation on the bus to signal invalidation (J. Goodman, personal comm. 1985).

Second, the *cost* of implementing each state must be weighed against the improvement in *performance* that it will offer in the system at hand. To illustrate, consider the use of the *clean source states* (read and write) as opposed to having just dirty source states. If a cache is the source of a block, it will provide that block when another cache requests it. Such a request to a cache will reduce the cache's availability to its own processor, thereby creating contention for the cache — contention between the processor requests and the bus requests. Since the block is clean, the latest version also resides in main memory, from where it could alternatively be fetched. However, suppose that a fetch from memory is much slower than a fetch from another cache. Then if the requester must hold onto the bus while waiting (because the address/data phases for a read are not split), the bus contention generated by fetching from memory can be worse than the contention for the cache generated by fetching from the cache. Also, if the requesting processor's cache does not implement prefetch, then a fetch from memory, as compared to a fetch from another cache, will entail a much longer wait for that processor — even if bus traffic is minimized by splitting the address and data phases.

In short, the clean source states are implemented only if fetching from another cache is sufficiently faster than fetching from main memory. If this is not true, then the read-source-clean state should be eliminated, using the plain read-state instead, and the write-source-clean state should be changed to write-clean.

### *B.2.2. Protocol in Action*

The state transitions will be illustrated with figures and a table now, clarifying the behavior of the caches and the rationale behind the state features. The full details of the processor requests, cache responses, and bus commands are deferred to Appendix 2. Throughout the discussion, the inventors of each feature will be cited.

**Figures.** The request-response sequences shown in the figures illustrate the interaction of the processors, caches, and memory. Keep in mind that the last cache to fetch a block becomes its source, and provides the block when the block is next requested by another cache (unless the source purged the block in the meantime).

**No Source.** Figures 12 and 13 show that if there is no source cache for the block, even if the block is present in another cache, the block is provided by memory. Furthermore, the requester cache assumes *read/write/lock* privilege for the block if the processor's request is *read/write/lock,* respectively. But if the request is for *read* privilege, any cache that has the block signals *hit;* otherwise the requester will assume write privilege, as described in the next paragraph. Under the Papamarcos and Patel (1984) scheme, any cache that has the block (that signals *hit*) is a potential source. These caches then

arbitrate and the winner becomes the actual source. The two approaches will be compared in Section B.2.3, Feature 8.

Figure 14 shows that if the request is for *read* privilege and the block is *not present* in another cache, on the other hand, the requester assumes *write* privilege, so that if its processor subsequently writes the block, a bus access will not be required in order to obtain write privilege. The main use of this feature is in fetching *unshared data*. Papamarcos and Patel (1984) introduced this feature, and Goodman has suggested using it as it is used here (J. Goodman, personal comm. 1985). Alternatively, as will be discussed in Section B.2.3 (Feature 5), fetching unshared data for write privilege could be determined statically (Katz et al., 1985).

**Source.** Figure 15 shows that there is a source cache for a block, the source provides the contents of the block, if requested, along with the clean/dirty/lock status of the block. The presence of this status on the bus signals the presence of a source cache. Figure 16 shows that if the requester cache already has a valid copy at a processor write, it only requests write privilege, not the block itself.

Censier and Feautrier (1978) suggested the idea of *direct cache-to-cache transfer* in the context of a system without full broadcast. While Goodman (1983) implemented cache-to-cache transfer in the context of realizing the rich capabilities of full broadcast.

**I/O Transfer.** An I/O transfer is the result of a request made by a processor that desires to have data transferred into or out of the memory-cache system. The processor has designated the memory area and, if the data is shared with other processors, has synchronized the request with the other processors. Consequently, if an I/O processor is reading or writing a block of data, no other processor is writing the data, and no processor has the data locked in its cache (except in the case of bugs).

In executing an *input* operation, an I/O processor will simply invalidate the block in all caches as it writes to memory, while in executing an *output* operation, the I/O processor will fetch the block for write privilege, thereby invalidating the block in all caches.[4]

For the sake of speed, *if an I/O processor has a cache,* it should not use the cache as an intermediary in an I/O operation, but instead should transfer directly between the bus and its private buffers. In this case, the I/O processor's cache must respond like any other cache in the system, providing the block if necessary, as well as invalidating it. So either the I/O processor should have its own *separate interface to the bus,* independent of its cache, for use in I/O transfers, so that its cache will respond as usual to the bus requests. Or else the I/O processor should have *two special read and write commands* for its cache, signaling I/O read and write, as opposed to normal read and write. The second design may add substantial complexity to the cache, both in terms of control and

[4] Another possibility is not to invalidate, and to update the caches on input. But the performance gain, if any, may be too small to warrant the cost of extending the protocol.

datapaths, but the first requires a separate interface just for I/O.

By the way, the reason that an I/O processor will probably have a cache in a multiprocessor system is that its processes must synchronize with processes on other processors. and it will probably be simpler and more efficient to do so using synchronized access to shared memory, rather than hardware signals. In this scenario, an I/O processor would receive *service requests* from a service-request queue, and would reply to the requests after servicing them, say by moving a sleeping process to a ready queue.

**Efficient Locking.** Figure 17 illustrates how busy-wait locking is efficiently executed. The *first block* of the atom is fetched for write privilege and locked until the entire operation is done. Conflicting access requests are serialized by the bus when a block is locked, and the cache supplies the target word as on a read instruction. Further, as shown in Figure 19, the unlock can occur at the final write to the block. So the lock instruction is a special processor read instruction, and the unlock instruction may be implemented as a special write instruction. An unencoded way of doing this is to devote a separate processor line to the function, which will be interpreted by the cache as *lock* on a read and *unlock* on a write.

If another cache attempts to fetch the atom during this time, it will request the *first block* and find it locked, and the locker cache will record that another cache is waiting, using the lock-waiter state (Figure 18). This will only require accessing the *bus directory* of the cache in which the block is locked since *lock-waiter*, as well as *lock*, status needs to be maintained only in the bus directory. Consequently, the access will be very fast and will not interfere with the processor's use of its cache — in contrast to the fetching of an actual data block containing a test-and-set lock bit. The requester cache, then, enters the block address in a special *busy-wait register* in the cache, thereby setting the stage for efficient busy wait.[5]

In short, locking involves the lock and lock-waiter states.

- *Lock state:* The first block of the atom is fetched and locked in the cache until the operation is done.
- *Lock-waiter state:* An unsuccessful request from another cache for a locked block changes the state from *lock* to *lock-waiter*.

*Process Switching.* Under the above protocol, if a process holding a busy-wait lock were to be switched out, it would have to be loaded back into the same processor because that is where its lock is. However, this is not a problem for the following reason.

> Under any locking scheme, especially busy-wait locking, it is important to preclude the switching of processes (or threads of control) while a lock is held, in order to

[5] An option here is to implement *hint read*. In this case, a cache can fetch a word from a locked block in another cache, and pass the word on to its processor. Setting of the busy-wait register, that is, waiting at all, could also be made optional, if desired.

avoid prolonging the time for which other processes may have to wait for access to the atom.

This is achieved by disabling the *appropriate interrupts* (not all traps), by not requesting I/O in the atomic section of code, and by avoiding page faults while executing the atomic section. Page faults are avoided by having the compiler insure that no atomic section or hard atom crosses a page boundary. Or else, in the case of a process queue with dynamically-allocated entries, the descriptor and the entries should be memory resident — as they are in Unix, where the entries are process control blocks (Peterson, Silberschatz 1985; Denning, Dennis, Brumfield 1981). This also gives another reason for disabling certain interrupts: if a process queue, like a ready queue, is locked and an interrupt handler would require accessing that queue, the handler would be unable to gain access.

The lock/unlock instructions are actually *lock-disable* and *unlock-enable,* each taking an interrupt-mask, as well as an address and a register operand. The lock-disable instruction disables interrupts only if the lock is obtained, and the proper use of these instructions is insured by the assembler or compiler.[6]

*Purging Locked Block.* There is a problem with the lock-state scheme in a cache whose set size is not large: *a locked block may need to be purged,* in which case the lock will vanish. So for a cache that is not fully associative, a *lock bit* should be implemented for the first block of the atom. It will either be a hardware tag bit on each memory block, or else a bit in the first cell of the block itself. The compiler already understands that atoms must start on block boundaries (Section B.1.4). So if the lock is a data bit, rather than a hardware tag bit, the compiler will also understand that this bit in the first block of a busy-wait atom is reserved for hardware use. If necessary it will reserve the entire smallest addressable-unit containing the bit.

Under this scenario, when a block is locked by the cache, the cache not only sets the block's status to *lock,* but also sets the block's lock bit. Consequently, if the block must be purged when locked, the lock will not vanish, for it will be flushed to memory. If another cache subsequently fetches the purged block to lock it, it will first test the lock bit. If the bit is set, it will understand that the block is locked, just as if another cache had informed it, and it will take the same action as it would otherwise take, and will not store the block. If the processor that locked the block requests further access to the block before unlocking it, this read or write will cause the block to be fetched and stored in the cache with lock-waiter status.

---

[6] Just the same, for the sake of exploration, it might be noted that process switching while holding a busy-wait lock could be allowed if the protocol were extended. Specifically, suppose there were a bus command Lock to be used when a processor seeks to lock a block. Then the Lock command would be refused if the requested block were already locked, whereas a Read or Write command on the bus would cause a locked block to be transferred to the requester cache and locked there. In this case, if a process holding a block were to wake up on another processor, it will get the locked block simply by reading or writing it, as usual. In addition, the Unlock command would unlock the block even if the block were in another cache.

When the processor that locked the block finally tells its cache to unlock the block, if the block is no longer in the cache, the cache will simply write the final word (which may accompany the unlock instruction) through to memory, clear the bit in memory, and simultaneously broadcast the unlocking on the bus. If the bit is a hardware tag bit or if an addressable unit of the block is devoted to the bit, the cache will not need to fetch the bit; instead, it can just write a zero to the bit.[7]

*Explicit* vs. *Implicit Locking.* Under *explicit locking,* an explicit indicator of the lock status is set to *lock* and *unlock* when appropriate, whereas under *implicit locking,* the entire cache or memory module containing the data is held under sole-access by holding its port (or ports) during the atomic operation. Implicit locking can be used for implementing a processor atomic read-modify-write instruction; however, it suffers from several serious constraints, none of which are present with explicit locking.

Specifically, the advantages of explicit over implicit locking include these.

- *Fine-grained locking:* Only the target atom is locked (under the cache lock-state scheme), not the entire cache or memory module, thereby allowing maximal concurrency in access to cache or memory.
- *Traps permissible:* Traps do not require aborting the atomic operation, as they do under implicit locking (although process switching should be precluded anyway, as explained above).
- *Multiple blocks and modules:* The atom can span several blocks and several cache or memory modules, rather than being restricted to just one — though spanning should be avoided where possible, just the same, to reduce the lock-holding time.

Furthermore, under the cache-state method of explicit locking, *explicit locking is as fast as implicit locking.* This is because a block can be locked on the first read to it and unlocked on the final write to it, so both locking and unlocking can be concurrent with

[7] Several technical points include these. (1) The "L" response in Figure 18 need be nothing more than the lock bit itself. (2) The lock value in memory is relevant only when the block is not in a cache. Consequently, that value needs to be updated only when the block is purged (of course), or if the lock is unlocked while the block is not in a cache. (3) Just the same, if the lock values are always updated in memory when changed, these values may aid in crash recovery. (4) If, contrary to the approach advocated here, the lock state were used, not to eliminate test-and-set but to implement test-and-set, and if that were the only use of the lock state, then purging a locked block would never be a problem, so the extra lock bit would not be needed. (5) If it is desired to insure *hardware protection* against software errors that attempt to read or write a locked block, a lock could be an entire processor I.D. rather than just a bit. For, as mentioned, process-switching is precluded while a busy-wait lock is held. (6) Two other possible solutions to the problem of purging a locked block are these: sufficiently large set size or associative lock registers. Specifically, consider the potential nesting of traps during program execution. At each nesting level (from zero to the maximum possible $n$) at most one block will be locked by that level under a busy-wait policy, in order to avoid long lock-holding times. In addition, since process-switching is disabled, the maximum number of blocks that will ever be locked at a time in a cache is $n$. Consequently, if the cache set size were $n+1$, a locked block would never need to be purged. Alternatively, the cache could have $n$ fully associative lock registers containing the needed information on each lock that is held. The problems, here, are determining $n$, and if $n$ can be determined, the possible large value of $n$.

necessary reading and writing of the data. In particular, this overlap will *always* occur for processor atomic read-modify-write instructions that could otherwise be implemented using implicit locking, since an atom for such an instruction must be contained entirely on one block. Therefore, cache-state locking should be used to implement *processor* atomic read-modify-write instructions, as well as *programmer/compiler*-implemented busy-wait locking.

*Simple* vs. *Complex Instructions for Atomic Operations.* One may wish to consider whether a single read-modify-write instruction is better for an atomic operation or whether several instructions are better. This is the issue of simple *vs.* complex instructions, or reduced-instruction-set *vs.* complex-instruction-set approach to processor design (Patterson 1985; Colwell et al. 1985). Here the issue is applied just to atomic read-modify-write operations. Without resolving the issue, I will point out several advantages of the simple-instruction approach.

- *Simpler instructions:* no atomic read-modify-write instructions, just ordinary instructions
- *Fewer instructions:* no atomic read-modify-write instructions, just a lock-disable instruction and an unlock-enable instruction (or two unlock-enable instructions, one that writes a word and one that does not)
- *Easy implementation of new atomic operations:* no need to microprogram; just write macro-code for a program or compiler to implement

In any case, whether the simple or complex instruction approach to atomic operations is taken, the cache-state method offers the most efficient locking and unlocking method.

*Zero-Time Locking/Unlocking.* Under the lock-state method, a separate block fetch for the lock bit is not required if the first block of the atom is to be read, since fetching that block is overlapped with fetching the lock. But in addition, separate cache accesses to lock and unlock an atom can be eliminated if the first read and the last write of an atom are to the first block of the atom, since locking and unlocking can then be overlapped with the first read and the last write, respectively. How often will these last two occasions for overlap occur (thus implying the first)? Let us consider this question in the context of each of the two reasons for busy wait, in turn.

- A situation where busy wait is *less costly* than sleep wait
- A system where busy wait is *necessary* to implement sleep wait

In a situation where busy wait is *less costly* than sleep wait, the atom is probably contained *entirely on one block* since this kind of operation is intended to be very fast, especially if implemented as a processor atomic read-modify-write instruction.[8] Just the same, the speed only needs to be better than that for implementing sleep wait. If sleep wait is implemented using busy wait (instead of P/V hardware, described in Section C.1), then it will probably entail several block fetches, as well as reasonably long wait times and processor operations. So in this case, a detailed analysis of the cost of sleep wait may justify using busy wait on an atom that is not contained entirely on one block.

[8] If an atomic read-modify-write instruction were implemented using implicit locking, the atom *must* be contained entirely on one block to avoid very complicated cache hardware, very low concurrency locking

But without such an analysis, *a reasonable rule of thumb* is to limit this operation to atoms that are contained entirely on one block, or sub-block transfer unit, so that data misses will not occur during the operation and thereby slow the operation down. In the case of sub-block transfer units, described in Section B.1.4, blocks will tend to be larger, so even if one wishes to allow data misses during busy wait by allowing an atom to span transfer units, the chances of fitting the atom entirely on one block are greater anyway. The *upshot* is that if the atom fits entirely on one block, the first read and last write to the atom will both occur to the first (and only) block of the atom.

In a system where busy wait is *necessary* in order to implement sleep wait, the atom is a process queue, probably dynamically-linked, as in Unix (Peterson, Silberschatz 1985) and in Denning, Dennis, and Brumfield (1981). In any queue, the first block of the atom contains the queue descriptor, and since a block probably contains at least two four-byte words, there will be enough room for a head and tail pointer, or one pointer and some additional information. The descriptor will always be read first. In addition, if the descriptor is written, it can be written last.[9] But if the descriptor is not written, the unlock must, indeed, cost an extra access.

To get a feel for whether the descriptor will be written, let us consider several queue implementations.

- *FIFO:* the descriptor will be written on both insertion and deletion
- *Priority — linear sorted list:* the descriptor will be written on deletion, but not necessarily on insertion
- *Priority — binary tree or heap:* the descriptor (which points to the tree root) may be written on insertion and deletion
- *Priority — array of linked lists, each list covering a priority range:* the descriptor will indicate the highest range that contains a non-empty list (for fast deletion), so may be written on insertion and deletion

We see, then, that under both occasions for busy wait, the first read and the last write of an atom will probably occur on the first block of the atom.

Having developed an efficient scheme to lock a block — and an efficient scheme for another cache to initially find out that the block is locked — our concern now turns to efficient busy waiting for a locked block.

---

(locking an entire module for a prolonged period of time), and the potential of deadlock among caches in such a state. The lock state, fortunately, avoids all of these problems.

[9] In some cases, the descriptor *must* be written last. If not, there is some chance of incurring a miss if it is written last, but this cost should be negligible since the number of blocks referenced by the processor during the time of locking should be small. Or from a high-level point of view, if the miss rate on the atom blocks just fetched is not negligible, the cache is probably not very effective overall.

**Efficient Busy Wait.** Three proposals for efficient busy wait will be considered.

- *Busy-wait register (introduced here):* wait for unlocking, then arbitrate; winner locks lock; losers continue waiting
- *Update-by-block (write-back): invalidate* other caches on a write, namely, when locking or unlocking a lock
- *Update-by-word (write-through): update* other caches on a write, namely, when locking or unlocking a lock

Keep in mind that although the first method is advocated here, the other two methods can be integrated with the lock state by modifying the lock-state protocol appropriately.

*Busy-Wait Register.* There are two purposes in efficient busy wait.

- Eliminate unsuccessful retries from the bus
- Relieve a waiting processor of polling the status of a lock, allowing it to work while waiting

When a process is busy waiting, it is running. However, it should not access the bus in order to retry, but should wait for the unlocking event to be broadcast on the bus, and then respond appropriately. Furthermore, since the process is running while waiting, it is best if it can do useful work during this time, instead of polling the lock status. This can be arranged by having the process *prefetch* the atom — requesting the cache to fetch and lock the first block just before the process is ready to use it. More generally, if a section of code can be specified such that the process is ready to access the atom while executing the code, then the process can execute this *ready section* while waiting for the interrupt from its busy-wait register.

In considering efficient busy-wait methods, one may challenge, "What is all the fuss about? Busy wait should not last that long, since long waits are implemented using sleep wait, not busy wait." On the surface this is a good point, but it is countered by the second of the two reasons for using busy wait:

- A system where busy wait is *necessary* in order to implement sleep wait

The manipulations of the sleep-wait and ready queues that must be accessed in order for the software to implement sleep wait may require several block fetches, say three or four, per queue. And in addition, there may be quite a few processes that access each queue, especially the ready queue, thereby generating high contention for the queue. Efficient busy wait addresses this situation. (On the other hand, efficient sleep wait, considered in Section C.1, eliminates the need for busy wait, along with the substantial time overhead incurred by that method.)

When a locked block is *unlocked,* this action is broadcast on the bus if the state in the locker cache is *lock-waiter* — indicating that another processor had requested the block while it was locked (Figure 19). A busy-wait register waiting on that lock recognizes the unlocking, and then initiates a *bus arbitration.* In this case, the winning cache will fetch the block for write privilege, lock the block using the lock-waiter state (since that will probably be appropriate), and interrupt its processor; while the other caches will let their

processors continue whatever they are doing and will not access the bus, making no attempt to fetch the block again (Figure 20).

Regarding the *bus arbitration*, after the unlocking, the next bus arbitration will give priority to those caches that are waiting for the lock. This can be easily implemented by having those caches specify very high priorities, say by devoting the most significant priority bit for this purpose — only the relevant caches will set it to a logical high. Then if it turns out that there are no waiters after all (because the waiting processors were switched out of their processors), the arbitration will proceed normally, with no wasted time.

This method of busy wait meets the two efficiency requirements fully.

- Eliminates all unsuccessful retries from the bus
- Allows a processor to work while waiting[10]

*Update-by-Block (Write-Back).* Under the standard update-by-block protocol for efficient busy wait, a block is invalidated in other caches when the block is written. Censier and Feautrier (1978) may have been the first to suggest this method of busy wait; while Sequent has implemented it in the context of an update policy that writes each word through to memory but invalidates other caches rather than updating them (*System* 1984; *Guide* 1985). (Also, under the Sequent system, memory, rather than the processor, executes the atomic test-and-set instruction.)

A version of this busy-wait method developed by Yale Patt (personal comm. 1985) is extended here by explicating mechanics for a processor atomic read-modify-write instruction in a cache. The result is that the number of unsuccessful retries on the bus is reduced to the minimum possible under this approach to busy wait, namely one retry for each waiting processor at each unlocking of a lock.

Under this method, when a processor accesses a lock bit to set it, its cache initially assumes *read*, not write privilege (shared access, not sole access), in case other processors are also waiting on the bit in their caches. Each processor waiting on that bit continues testing its copy in its cache until the block is *invalidated*. That is, when the bit is to be cleared, the processor that set the bit no longer has write privilege (sole access) to the block, so it must go to the bus and get write privilege, i.e., invalidate the block in the other caches. Each waiting processor, therefore, will automatically generate a *miss* at its next test of the bit, so its cache will fetch the block for read privilege. However, the miss should initiate an atomic read-modify-write action under which the cache will *hold the bus* while the processor tests the bit. If the bit is zero, it will be updated to one; otherwise the instruction will just abort, and the processor will resume testing as before. Each waiting processor after the first to acquire the bus will therefore read a value of

[10] Note, just the same, that if one did not want to allow for the possibility that the processor may have work to do while waiting, the processor could, alternatively, poll the busy-wait-register signal.

*one*, so will continue testing as before.

The hardware complication, here, is that there must be *a special read-modify-write protocol*, distinct from the normal protocol, just to implement efficient busy wait. Normally a read-modify-write protocol takes effect at the read whether there is a hit or miss: the block is fetched for write privilege at the read, or else the bus is acquired at the read and is held through to the write, at which time the block is fetched for write privilege (Section B.2.3, Feature 6). Both of these normal protocols, however, subvert efficient busy wait. Hence there must be a special protocol for busy wait, under which the cache initiates an atomic read-modify-write action only if the read generates a miss, as above.

The two performance advantages of the busy-wait register scheme, given earlier, imply corresponding disadvantages of the standard update-by-block scheme. The *first disadvantage* is that at every unlocking all processors waiting on the lock, except one, generate unsuccessful retries at each unlocking, each retry requiring a block (or transfer-unit) fetch. So if the mean number of processors waiting on the lock is $n$, then the mean number of unnecessary block (or transfer-unit) fetches at each unlocking is $n-1$.[11] The *second disadvantage* is that a waiting processor must continually test the lock bit, thereby wasting its time if it has useful work to do.

*Update-by-Word (Write-Through).* Rudolph and Segall (1984), along with the DEC Firefly and the Xerox Dragon (reported by Archibald and Baer 1985), implement versions of the update-by-word scheme. This approach allows efficient busy wait by updating other caches with the new values of the lock as they are written, thereby removing the need for each waiting processor to access the bus and fetch the entire block (or transfer unit) when the lock is unlocked. (The reader may wish to review the issues regarding the two update policies, as presented in Sections B.1.1 and B.1.4.)

*Rudolph and Segall* orient their entire protocol toward efficient busy wait. Specifically, when any block is *first written* by a processor, the write goes through to memory and to all caches, *updating* all caches that contain the address. Whereas, when a block is written a *second* time (or more generally, the $n$th time for some $n$), the block is *invalidated* in other caches. This implies that when a lock bit is *cleared (unlocked)*, it will be the *second write* to the block, so the block will be invalidated in all other caches. The first

[11] Suppose, in contrast, that the special atomic read-modify-write protocol introduced above were not used; instead a processor tests the bit and if it is zero *then* initiates an atomic read-modify-write action under a normal read-modify-write protocol. In this case, each waiting processor must make two bus accesses at an unlocking, the first for the initial test, which will probably read a zero, and the second for the read-modify-write, which may read a one since another waiting processor may have set the bit in the meantime. The first access will require a block (or transfer-unit) fetch, and the second will require the same — except for the processor that sets the bit, which will simply require invalidation. Hence the mean number of unnecessary block fetches will be $2(n-1)$ instead of the $n-1$ required by the special read-modify-write protocol given above. Just the same, under the special protocol, each waiting processor must hold the bus while testing the bit, and this will take some time if it cannot be overlapped with the fetch of the rest of the block (or transfer unit). If this time approaches that of the arbitration and fetch, then the special protocol is of little value.

processor to fetch the block, then, will make the *first write* to the block (using an atomic read-modify-write) to *set (lock)* the bit and will update all caches of processors waiting on that lock, allowing those processors to continue busy-waiting in their caches without accessing the bus.

An atomic read-modify-write instruction, in this scheme, is identified as such to the cache and goes through to the bus as such, maintaining sole access to a memory unit, using explicit or implicit locking on that unit. Because of this, all unsuccessful retries are eliminated from the bus using the foregoing protocol.[12]

The *DEC Firefly* and *Xerox Dragon* implement the following innovative update policy.

- *Update-by-block (write-back):* for unshared blocks (copies in no other caches)
- *Update-by-word (write-through):* for shared blocks (copies in other caches)
  - Update other caches: Firefly and Dragon
  - Update main memory: Firefly only

Both schemes use the bus *hit* line (described earlier) on bus *writes*, as well as on bus *reads*, to detect shared/unshared status for a block. Section B.1.4 presents further discussion of the general performance of this scheme.

**Summary.** Efficient methods of busy-wait locking, waiting, and unlocking have been introduced.

- Efficient locking
  - *Time cost:* probably zero — the lock and the first block of the atom are fetched concurrently; locking is also concurrent with the first read of the atom if that read is to the first block (as it probably is)
  - *Memory cost:* little or none — a free cache state, and if necessary, a hardware tag on each block, or one bit (or addressable unit) of a busy-wait atom
  - *Control cost:* the cache bus control-unit must lock blocks; the bus must have a code for *locked*
- Efficient waiting
  - *Time cost:* little or none — no unsuccessful retries; work while waiting (if possible)
  - *Memory cost:* associative busy-wait register in each cache
  - *Control cost:* a match in the busy-wait register motivates bus arbitration, and if won, a processor interrupt

[12] Rudolph and Segall also have block fetches update any cache in which the block has invalid status. However, this is not necessary for efficient busy wait, so the motivation for the feature is not clear to me. They also specify one-word blocks, probably to minimize internal fragmentation due to the devotion of one block to each busy-wait lock, as discussed in Section B.1.4.

- Efficient unlocking
  - *Time cost:* probably zero — unlocking is concurrent with the last write to the atom if that write is to the first block (as it probably is)
  - *Bus access:* only if needed because of a waiter
  - *Control cost:* the cache bus control-unit must unlock blocks; the bus must have a code for *unlocked*

Finally, as stated earlier, although the busy-wait register method of waiting is advocated here, the other two waiting methods (update-by-block and update-by-word) can alternatively be integrated with the lock state by modifying the lock-state protocol appropriately.

**Table.** The preceding examples illustrate the capabilities that can be given a broadcast cache protocol, while Table 1 summarizes the corresponding state transitions for a block. In order to avoid cluttering the table, the following transitions are omitted.

- *Source:* A cache that fetches a block becomes its new source.
- *Clean/Dirty:* When a processor writes a block, the block becomes dirty (if not already dirty).
- *Lock Waiter / Busy Wait:* When a locked block is requested by another cache, the status becomes *lock-waiter* in the cache holding the block, and the requester's *busy-wait register* is loaded.

Table 1a, for example, shows that if a cache places a read request for a block on the bus and the status for the block in another cache is write, the new status for the block in the requester cache becomes read. Table 1b shows that for the same situation, the new status for the block in all other caches containing the block (including the source) is read. Since the source had write privilege for the block in this example, though, it alone had a copy of the block in this case. Appendix 2, finally, offers fine details of the requests and responses in the protocol.

**Table 1. State Transitions for Cache Block**

**Table 1a.**

| | New Status in Requester Cache | | | |
|---|---|---|---|---|
| *Cache request to bus* | *Initial Status in Other Cache* | | | |
| | Invalid | Read | Write | Lock |
| Read | Write[1] | Read | Read | Invalid |
| Write | Write | Write | Write | Invalid |
| Lock | Lock | Lock | Lock | Invalid |

**Table 1b.**

| | New Status in Other Cache | | | |
|---|---|---|---|---|
| *Cache request to bus* | *Initial Status in Other Cache* | | | |
| | Invalid | Read | Write | Lock |
| Read | Invalid | Read | Read | Lock |
| Write | Invalid | Invalid | Invalid | Lock |
| Lock | Invalid | Invalid | Invalid | Lock |

**Note**

1. Write privilege is assumed if the block is invalid (or absent) in *all* other caches; otherwise read privilege is assumed.

### *B.2.3. Protocol Evolution*

Now let us explore the evolution of broadcast update-by-block (write-back) protocols. Table 2 traces key steps of this evolution, and has two parts, the upper part showing the evolution of states, and the lower showing the evolution of other features. The states and the features will be discussed in turn, following a presentation of overall perspective. Keep in mind that most of the features were discussed in Section B.2.2, to which the reader may return when helpful.

**Overall Perspective.** The information on the *classic approach* (column 1) is taken from Censier and Feautrier (1978) who give no reference to the literature in this regard, so their word is accepted at face value. The classic approach does not necessarily implement update-by-block (write-back), but it does implement a form of broadcast, along with dual directories. Specifically, a write action in a cache is broadcast on a high-speed bus to all other caches, which will then invalidate the corresponding block if it is valid there. This scheme does not, however, guarantee that conflicting access requests for single reads and writes (to hard atoms) will be serialized. The classic approach uses *identical dual directories* to eliminate the interference of irrelevant requests.

The other schemes shown in Table 2 follow the *dual-directory approach*, (Feature 3) although Goodman (1983) and Frank (1984) independently reinvented it (J. Goodman, personal comm. 1985). Goodman also implemented the *source* function, or *direct cache-to-cache transfer*, under an update-by-block (write-back) policy (Feature 1). In this case, one cache provides the data directly to another cache that requests it, if the first cache has the latest version and has not yet updated main memory. Censier and Feautrier had suggested this as a possibility, but Goodman was evidently the first to implement it, and Frank's protocol also makes it possible. Papamarcos and Patel developed their scheme from Goodman's; Katz evolved their scheme from Frank's and Goodman's; while I attempt to identify the most promising features of each, as well as add new innovations.

Table 2. Evolution of Broadcast, Update-by-Block (Write-Back), Cache-Synchronization Schemes

| **States** (*read* = *shared*-access privilege; *write* = *sole*-access privilege)[0] | Classic[1] (pre-1978) | Goodman (1983) | Frank (1984) | Pap,Patel (1984) | Katz et al (1985) | Current proposal |
|---|---|---|---|---|---|---|
| | *(Regarding states: N = non-source state; S = source state)* | | | | | |
| Invalid | N | N | N | N | N | N |
| Valid | N | | | | | |
| Dirty (for update-by-block) | N | | | | | |
| Read[2] | | | | | N | N |
| Read, Clean | | N | N | S[3] | | S[3] |
| Read, Dirty[4] | | | | | S | S |
| Write, Clean | | N | | S[3] | S[3] | S[3] |
| Write, Dirty[5] | | S | (S)[6] | S | S | S |
| Lock, Dirty[7] | | | | | | S |
| Lock, Dirty, Waiter[8] | | | | | | S |
| **Features** | | | | | | |
| **1. Update-by-block; dual control units; cache-to-cache transfer; serialization of conflicting single reads and writes** | | ✓ | ✓ | ✓ | ✓ | ✓ |
| **2. Fully-distributed state information:** valid / read / write / lock / dirty / source status (V/R/W/L/D/S) *(faster response of caches; greater consolidation of state information; simpler memory)* | VD | RWDS | RWD | RWDS | RWDS | RWLDS |
| **3. Directory Duality:** 2 Identical Dual (ID) / 2 Non-Identical Dual (NID) / 1 Dual-Ported-Read (DPR). *(DPR reduces the hardware; NID eliminates interference due to updating status — dirty status is only in processor directory, waiter status is only in bus directory)* | ID | ID | ID | ID[9] | DPR[10] | NID[11] |
| **4. Bus invalidate signal:** no need for invalidation write-through<br>**On write hit:** Gain write privilege with a one-cycle invalidation *(instead of a word-write to memory)*<br>**On write miss:** Gain write privilege while fetching block *(instead of a word-write to memory)* | | | ✓ | ✓ | ✓ | ✓ |
| **5. Fetching data for write privilege on read miss:**<br>**Unshared data:** unshared status is determined statically(S) or dynamically(D) *(save bus arbitration and invalidate cycle if the data is subsequently written)* | | | | D | S | D |
| **Shared soft atoms:** if the atom will probably be written | | | | | | S[12] |
| **6. Processor atomic read-modify-write instruction:** serialize conflicting access requests (✓); also allow efficient busy wait (E) | | | ✓ | ✓ | ✓ | E |
| **7. Flushing on cache-to-cache transfer:** flush block (F), or do not flush block (NF); transfer clean/dirty status with the block (S) *(F is desirable unless bus and memory do not support it, in which case it would slow down the transfer; NF requires transfer of clean/dirty status if block may be clean or dirty — see source states above)* | | F | NF | F | NF,S[13] | NF,S[14] |
| **8. Number of sources for read-privilege block:** allow *multiple sources*, thus a source for a read-privilege block must *always arbitrate* before providing the block (ARB); allow *loss of (single) source*, forcing the block to be *fetched from memory* (MEM); have last fetcher become source, allowing LRU replacement across caches (LRU) | | | | ARB | MEM | LRU, MEM |
| **9. Writing without fetch on write miss** *(no block fetch when saving process state or initializing entire block)* | | | | | | ✓ |
| **10. Purging dirty read-privilege block:** No flush if block in other cache *(save block write)*; irrelevant under Option F, Feature 7 | | | | | | ✓ |
| **11. Memory mode** *(system analysis; reliability)* | | | | | | ✓ |
| **12. Efficient busy wait** *(no unsuccessful retries on bus; process can work while waiting)* | | | | | | ✓ |

**Table 2 Notes**

0. The status concepts appropriate for update-by-block (write-back) are not the same as for update-by-word (write-through), although there is some correlation.
   - Update-by-block (write-back)
     - Block status indicates *processor access-privilege* to the block
     - Read privilege = shared-access privilege
     - Write privilege = sole-access privilege
   - Update-by-word (write-through)
     - Block status indicates the block's *residency among the caches*
     - Shared residency, among several caches
     - Sole residency, in one cache

   Under update-by-block, *write privilege* implies *sole residency*, but *read privilege* does not imply *shared residency*. Conversely, *shared residency* implies *read privilege*, but *sole residency* does not imply *write privilege*. Just the same, as shown in Feature 5, sole residency detected on a block fetch can be used to assume write privilege.

   On the other hand, under update-by-word, a processor can write any valid block. But if main memory does not need to be updated, as in the Firefly and Dragon protocols (Section B.2.2), then a write to an unshared block need not go through to the bus.

   A convenient approach to this terminology may be to use *shared* and *sole* for both write-back and write-through schemes, in spite of the fact that they would have different meanings under the two schemes. On the other hand, this approach may generate more confusion than convenience.

1. Referred to by Censier and Feautrier (1978), who give no reference to the literature in this regard.
2. This is any read-shared copy of a block other than the source copy.
3. Source function for a clean block is useful only if fetching from another cache is significantly faster than fetching from memory.
4. The dirty read state is useful only if a block is not flushed when transferred, i.e., Option NF of Feature 7.
5. The dirty write state offers the most impelling reason for the source function: the current cache has written the data, so under update-by-block, memory does not have the the latest version.
6. A source cache provides data only for a write request, not a read request. I have also heard that Synapse may not actually implement cache-to-cache transfer here even though the protocol described in the article makes it possible.
7. The lock state eliminates the time cost in busy-wait locking and unlocking.
8. The lock-waiter state indicates another cache is probably waiting, so the unlock should be broadcast on the bus.
9. No specification is actually given as to whether the directories are identical or not. ID is inferred because they cite Goodman and do not state that the two directories are different.
10. The cache data-store is also dual-ported read.
11. NID will be implemented if the performance gain appears worth the cost.
12. I point out the possibility.
13. The need to transfer clean/dirty status can be eliminated by changing the clean write-state to a non-source state. This also eliminates an inconsistency in the protocol, as discussed in the text.
14. Option F should be implemented in a system where it does not slow down a transfer significantly. Option NF,S is depicted in order to explicate the option having the more complex protocol.

**States.** From the table we see that Goodman was the first to implement the *source* function, with the *dirty write* state in which case the cache, not memory, has the latest version of the block. This was subsequently extended by Papamarcos and Patel to include the *clean* (read and write) states, which are useful if fetching from another cache is sufficiently faster than fetching from memory. Katz et al. introduce the *dirty read* state, which is useful if a dirty block cannot be flushed to memory at high speed while it is transferred to another cache. The current proposal, finally, introduces the *lock* states.

*Goodman.* Under this protocol, a cache becomes the *source* of a block only when it has *the latest version* (dirty status), which occurs in this protocol only when a cache has written the block twice. Specifically, when a dirty block is transferred from one cache to another, it is also flushed to memory, so it arrives clean. In addition, the first write to the block goes through to memory and invalidates the block in all other caches — since the original Multibus does not allow an invalidation signal — so the block still remains clean. The block becomes dirty only on the second write, at which time the cache becomes the source of the block. The costs of the invalidation write-through and the flush to memory will be considered in the discussions of Features 4 and 7, respectively.

*Frank.* The Synapse computer has its own proprietary bus, which enables invalidation concurrent with block fetch (Feature 4). Consequently, the *clean write* state is not useful here, as it is under Goodman's protocol.

*Papamarcos, Patel.* This scheme introduces the *clean write-state,* which is useful for *fetching unshared data on a read miss,* since no other process will be using the data (Feature 5). If it subsequently turns out that the block is not written, it will not need to be flushed to memory when purged. In addition, although

Papamarcos and Patel do not consider this option, if the block is not written, the cache will not need to provide the block, which is advantageous if fetching from another cache is not significantly faster than fetching from memory.

Papamarcos and Patel do not consider this option, for under their scheme, if a cache has a block, it also has source responsibility for the block. This extends the *source* function from dirty to *clean* states, which is useful *only if fetching from another cache is significantly faster than fetching from memory,* as stated in Section B.2.1.

More specifically, a cache will be designed with two interface registers, one for the bus and one for the processor, and can also be given dual-ported read capability, as in Borriello et al. (1985). Each register holds the contents of a block, so a fetch from the bus to a cache will delay the cache's processor for at most one cache read-cycle — time to read the block into the bus register. Furthermore, according to Smith(1982), cache access time is five to ten times shorter than memory access time, although this may not hold for lower cost, microprocessor systems. Consequently, if a requester must hold onto the bus while waiting for a memory read, the clean source-state will reduce the bus use-time by a factor of about five to ten, while delaying the processor of the source cache at most one

read cycle. In addition, if the requester cache is not prefetching, the processor wait-time will be reduced by the same factor of five to ten.

*Katz, Eggers, Wood, Perkins, Sheldon.* This scheme introduces the *dirty read* (source) state. The write-dirty-source state is converted to *read-dirty-source* in a cache when another cache requests read privilege for the block. The reason that the block remains *dirty* is because it is not flushed when is transferred, as it is in the Goodman and the Papamarcos and Patel protocols. The reason for *not flushing* the block is that if the bus or memory does not support the concurrent flush at all, or at the speed of the caches, the flush will slow down the cache-to-cache transfer or require an extra transfer to memory. (The flush is discussed further in conjunction with Feature 7.)

Regarding the transfer of *source* (and dirty) status, the requester cache assumes the plain read state, since the source status is not transferred at this point. Instead it is transferred only when another cache requests write privilege, in which case it obtains write-dirty-source status — or write-clean-source status, if the block is clean. The new protocol, in contrast, transfers source status here. The reason for transferring source status is that it will implement a *least-recently-used replacement algorithm across caches* — if the last cache to fetch a block tends to be the last to purge it. In this case, the chance of losing a source for read-shared blocks is reduced (Feature 8).

I feel that it is instructive to point out what appears to be an inconsistency in the Katz et al. protocol that is due to the *ownership terminology* — which they borrow from Frank. They extend Frank's protocol, along with the ownership terminology, but the terminology seems to obscure the three independent status categories (invalid/read/write, clean/dirty, non-source/source). Specifically, the terminology requires that if a cache has *write* privilege for a block, it also have *source* responsibility. Consequently, in order to get a *clean* write-state for fetching unshared data at read misses, they are forced to make the state a source state. What they really desire is a non-source clean write-state, as in Goodman's protocol. And with the proper terminology, they can achieve this end. Unfortunately the source status on the clean write-state generates a side effect that has some cost: clean/dirty status must now be transferred (along with the data) in a cache-to-cache transfer, since a source may now have either a clean or a dirty block. This issue is further discussed with Feature 7.

*Current Proposal.* This scheme is the result of identifying and carefully analyzing the features of the other protocols, as well as introducing several new states and features. The table-notes, along with this discussion, are intended to *clarify the reasons for each state and feature* so that a designer can select those that look most promising for their system, and can then follow through with a detailed performance and cost evaluation of those features in the context of their system.

In view of this, the current proposal has both *clean and dirty source* states, each useful for the reasons already discussed. In addition, the current proposal introduces the *lock* (dirty, source) state. This state carries the concept of state information beyond

read/write privilege to that of lock privilege, and distributes its location and control among the caches, continuing the evolutionary trend of full-broadcast cache protocols (Feature 2). This allows the time cost of locking and unlocking to be eliminated, since locking occurs concurrently with a read, and unlocking can occur concurrently with a write. Finally, the *lock-waiter* state is proposed, which informs the cache when it must broadcast the unlocking of a block on the bus, namely, if another cache requested access to the block when it was locked.

**Features.** Keep in mind that a feature may speed up a particular operation, but overall system speedup can only be determined from estimating the frequency of the operation in the system. If the frequency is too small, then the feature will not pay off.

*Feature 1.* These features were discussed above. Note that the serialization of processor atomic read-modify-write instructions, which moves beyond single reads and writes, is Feature 6.

*Feature 2: Fully-Distributed State Information.* The advantage of fully distributing the state information is that it enables a cache to respond quickly to requests, it is consolidated in just a few bits per block frame ( $\lceil \log_2 \#states \rceil$ ), and it simplifies the structure of memory. Frank, however, does not fully distribute the *source* status, maintaining a *source bit* in main memory, which indicates whether memory is the source or not. But following Goodman's innovation, in a system with fully-distributed source status, if a cache is the source, it informs memory not to provide the data when the cache services a bus request. The current proposal, in addition, distributes *lock* status, as well as invalid, read, write, dirty, and source status.

*Feature 3: Directory Duality.* Goodman and Frank reestablished the classic approach of identical dual directories, and Katz et al. introduced a single, dual-ported-read directory, which reduces the directory hardware (Borriello et al. 1985).

However, under both schemes, interference between the bus and processor accesses to the cache will be generated whenever the processor writes to the cache, for the status of the written block must be updated to *dirty* at this time. Bus requests will be bombarding every cache continually, so the bus directory (or single directory) will be very busy due to bus requests. Furthermore, according to Smith (1985), the frequency of writes may vary from 5% to 35% of a processor's memory references. Consequently, one may want to reduce, or eliminate, this interference.

Two methods of *reducing* this interference are to update the dirty status only when it changes; or else in a lower performance design (Borriello et al.), have the read and write cycles alternate. Another option is to *eliminate* the interference *entirely* by having *non-identical directories.* In this case, only the processor directory maintains clean/dirty status. This information is accessed by the cache's bus-controller only when the cache data is accessed, in which case interference with the processor must occur anyway.

Accordingly we ask, How much smaller is the frequency of changing a block dirty-status than the total write frequency? Appendix 3, Section 4, derives a formula for this frequency — the frequency of changing a block dirty status to 'dirty', or equivalently, the frequency of a write hit to a clean block. Estimates of .2% to .75% are derived; so if this range is representative, the interference from updating dirty status to 'dirty' when it changes appears to be negligible, not warranting non-identical directories on that ground alone.

Under the current proposal, we might point out, non-identical directories eliminate not only the interference of updating *dirty status* by the processor (which appears negligible at this point), but also eliminate the interference of updating *lock-waiter* status by the cache's bus-controller, as discussed in Section B.2.2.

*Feature 4: Bus Invalidate Signal.* Under Goodman's protocol, a cache obtains write privilege to a block (invalidating the block in other caches) on the processor's first write to the block by *writing through* to memory. As mentioned earlier, this is because the original Multibus, for which Goodman was designing, does not give the designer enough flexibility to explicitly signal invalidation (Section B.2.1). Later protocols make the assumption, in contrast, that the bus does allow *explicit invalidation*. On a *write miss* the invalidate signal allows invalidating while reading the block. While on a *write hit* to a block for which the cache has only read privilege, the same signal allows a *pseudo-write (or pseudo-read)* that invalidates the block in other caches (and in Frank's protocol, clears the source bit in memory), but does not initiate a memory cycle; thus it can be limited to one bus cycle even if a memory cycle takes longer.

Just the same, *if a cache gains write privilege only at a processor write* (Feature 5 is not implemented), then the cost of the write-through in terms of bus traffic is small if the cache blocks are reasonably large, say $n$ bus words. This is because the extra bus traffic appears to be much less than $1/n$ under typical access patterns, as shown in Appendix 3. In addition, the initial write-through is an *advantage* if it happens to be the only write to the block and the block is purged before being fetched to another cache. This is because the block is still clean after the write of the *first word*, so will not require a flush of the *entire block* when purged. Also, if the block is fetched to another cache before being purged, there is no advantage to the write-through in a scheme in which cache-to-cache transfer of a dirty block is as fast as cache-to-cache transfer of a clean block.

Regarding the extra bus traffic generated by the invalidation write-through, an *upper bound* on this cost can be derived from an impossibly bad case. Specifically, there are two occasions for block invalidation at a processor write: a *hit* on a block for which the cache only has read privilege, and a *miss*. In the first case, any protocol requires that the cache access the bus to obtain write privilege, whereas in the second case, only Goodman's protocol requires an extra bus access to obtain write privilege. Consequently, the relative cost of the invalidation write-through, as compared to invalidation without the write-through, is greatest on write misses. *So an impossibly bad case* for the invalidation write-through scheme is that *every miss is a write miss*, thereby requiring an

invalidation write-through at the time of every block fetch. If, further, the invalidation *write* is instead assumed to be a *read* (which may actually take longer than a write), the invalidation simply increases the block size by one word. The resulting bus-traffic comparison ratio is $(n+1)/n$, so an upper bound on the fractional increase due to the invalidation write is $1/n$. For example, if $n$ is 8, the upper bound is 12.5%

*Feature 5: Fetching Data for Write Privilege on Read Miss.* The last three protocols allow a block to be fetched for write privilege at a read miss in order to *fetch unshared data*. This does not reduce concurrent access to the data since the data is unshared; and if the data is subsequently written by the processor, the bus will not need to be accessed at that time in order to gain write privilege.

Papamarcos and Patel introduced the fetching of unshared data for write privilege by using a *dynamic* determination of whether the data is being shared or not, namely, whether some cache currently has a valid copy of it or not. This simply requires an open collector *hit* line on the bus. If a cache is purged at a process switch, this scheme will guarantee fetching all unshared data for write privilege at read misses. But if a cache is not purged at a process switch, when a process is later run on another processor, it is possible that some of its writable data will still be in the first cache. However, it intuitively seems that this chance would be fairly small since quite a few processes would probably have been run in the meantime and have wiped out most, if not all, of the data of the earlier process. In fact, if this is not true, then one might argue that the caches are larger than needed for the granularity of the processes.

Katz et al., on the other hand, suggest a *static* determination of unshared status, which is somewhat more complicated. *First,* the processor must have a special instruction to read data for write-privilege, which will apply only if the access is a *miss*. *Second,* the user must be able to inform the compiler, through type declaration, which data objects may be *read-shared among processes,* so that all other data objects will be fetched for write privilege on read misses. Data objects that may be read shared among processes consist of three kinds: read-only objects, hard atoms (if they are ever concurrently read by several processes), and soft atoms that are reader/writer synchronized by the software (to allow multiple readers). If these three kinds of objects are distinguished as such, then all other data objects should be fetched for write privilege on read misses. This would be the optimal static-determination approach for fetching unshared data.

However, since the dynamic approach to fetching unshared data appears intuitively adequate and much simpler, *static* determination can, instead, be used to fetch *shared* data for write privilege on read misses, namely for soft atoms that are written during any access session. For example, an access to a producer/consumer buffer will nearly always write the buffer descriptor (unless the buffer is full/empty on an insert/delete). Yet the first access to the descriptor will read it. The compiler could be given the intelligence to figure this out, and cause the data to be fetched for write privilege on a read miss.

Finally, as with Feature 4, an *upper bound* can be derived for the extra bus traffic

generated by a protocol that does not fetch unshared, writable data for write privilege at a read miss. We first assume that all data is unshared, implying that invalidation will generate no extra bus traffic under a protocol supporting both Features 4 and 5. We further assume that all misses are read misses and that every block is eventually written, thereby forcing another protocol to eventually invalidate every block at the time of the first write to the block. For another protocol, every block fetch will eventually be followed by an invalidate operation.

In this context, let $n_{arb}$, $n_{fetch}$, $n_{inval}$, $n_{read}$, and $n_{lat}$ be the number of bus cycles required for bus arbitration, block fetch, block invalidation, word read, and fetch latency (for internally-interleaved memory), respectively, and let $n_{words}$ be the number of words in a block. Then the cost-comparison ratio is $( n_{fetch} + n_{arb} + n_{inval} ) / n_{fetch}$, making the fractional increase $( n_{arb} + n_{inval} ) / n_{fetch}$. If every word-read requires an arbitration (a very-low-performance bus model), then $n_{fetch} = ( n_{arb} + n_{read} ) n_{words}$, while if the bus can be held for a multiword transfer (the usual bus model), then $n_{fetch} = n_{arb} + n_{read} n_{words}$ for non-interleaved or externally-interleaved memory, and $n_{fetch} = n_{arb} + n_{lat} + n_{read} n_{words}$ for internally-interleaved memory. (For a block read, an externally-interleaved memory requires the processor to issue an address sequentially to each module in turn, while an internally-interleaved memory distributes the initial address to the modules in parallel.) Since $n_{inval} \leq n_{read}$, we can assume equality and get an upper bound.

These assumptions imply that an upper bound for the very-low-performance model is $1/n_{words}$, the same as the earlier upper bound. Under the usual bus and memory models, if bus arbitration is overlapped with the prior transfer, as on the Multibus, then $n_{arb}$ is zero, again making $1/n_{words}$ an upper bound. More generally, if $n_{inval} = n_{read}/c$, then the upper bound decreases to $1/cn_{words}$. For example, for externally interleaved memory, $c = 2$, so if $n = 8$, the upper bounds are 12.5% and 6.25%, respectively. The effect of $c$ is identical in the upper bound for Feature 4 if the formula there is expanded and if arbitration is again overlapped. Also, for internally-interleaved memory, $n_{write} = n_{read} = 1$, so with overlapped arbitration, $1/n_{words}$ is again an upper bound. Appendix 3, Section 4, finally, derives much smaller, approximate upper bounds.[13]

*Feature 6: Processor Atomic Read-Modify-Write Instruction.* There are several ways to implement processor atomic read-modify-write instructions on (hard) atoms so that conflicting access requests are serialized. Let us consider four methods, the first of which requires going through to memory, while the others do not.

The *first* method modifies the update-by-block (write-back) policy, requiring a read-modify-write instruction to *access and hold a main memory unit as usual* (using explicit

[13] For Katz et al. $n_{words}$ is 8, making 12.5% an upper bound, yet the values they report are much greater, namely 12% to 32%. This is because they counted any bus access, whether for block invalidation, fetch, or flush, as generating the same amount of traffic (S. Eggers, personal comm. 1985).

or implicit locking), even if there is a valid copy of the block in the cache (Rudolph and Segall 1984). This requires that the processor inform the cache of the start of a read-modify-write instruction, it requires the cache to manage the bus appropriately and cache the block, and it requires that the block be invalidated in all caches containing the address — or updated in those caches in order to implement efficient busy wait.

Some processors do not signal the beginning of an atomic read-modify-write instruction, such as the Motorola 68000 — which has only test-and-set (*MC68000* 1982). However, this signal could be achieved by the programmer by writing a special register in the cache before and after the instruction execution; and the cache, or interface hardware to the cache, must be designed to interpret the register accordingly — as applying to the data references thereby encompassed. In fact, this is similar to the Intel 8086 implementation of an atomic read-modify-write, which is accomplished by a special bus-lock instruction prefixed by the programmer to every instruction in an atomic sequence (Rector, Alexy 1980).

The *second* method, that of Frank and of Katz et al., requires that the atom be contained entirely on one block, that *the block be fetched for write privilege at the beginning of the read-modify-write instruction,* and that the cache (or cache module) be held throughout the operation (implicitly locked). This approach again requires that the processor inform the cache at the start of the instruction, and it requires the cache to respond as mentioned. Papamarcos and Patel propose a variant of this: if the cache does *not* have *write privilege* for the block at the beginning of the operation, *the bus is gotten and held through to the write,* at which time write privilege for the block is obtained as usual. I do not see any advantage in this special case, over that of fetching the block for write privilege at the beginning of the operation, while the disadvantage is that the bus is held longer than needed. In any case, the second method, a standard update-by-block (write-back) method, does not allow the most efficient busy wait, as discussed in Section B.2.2.[14]

The *third* method for implementing a read-modify-write instruction again requires the processor to inform the cache of the start of the instruction. In this case, however, *the cache does not fetch the block for write privilege until the write,* nor does it hold the cache or bus in the meantime. Specifically, if the write generates a *miss,* it means that the block was stolen between the read and the write, so atomicity is violated, and the cache raises an *exception* that causes the processor to abort the instruction, and the cache accordingly aborts the pending write request.[15] Like the second method, this one does not allow the most efficient busy wait.

---

[14] The read-for-write-privilege instruction of Katz et al. will not, in general, work for initiating an atomic read-modify-write instruction. This is because it is designed for fetching unshared data, so applies only on *misses;* whereas for an atomic read-modify-write instruction it must apply on *hits* as well. This can be remedied if the software never just reads a hard atom, but always writes it (to initialize) or read-modify-writes it (to operate), in which case a hit implies write privilege. However, this constraint does rule out efficient busy wait, which requires fetching the lock bit for read privilege, to test it until it is found to be zero.

[15] Note that the cache cannot implement atomicity here by preventing the block from being stolen (or

Notice that if one did not want to build the intelligence into the cache itself for raising the exception, it could be built into the *processor interface hardware*, and the cache could simply signal this hardware on *every miss*. It might also be noted that every processor probably has some externally-generated exception that could be used here, such as the non-maskable interrupt on the Intel 8086, and the bus error exception on the Motorola 68000 (Rector, Alexy 1980; *MC68000* 1982)

The *fourth* method for implementing atomic read-modify-write instructions is to use the cache lock-state to realize explicit locking, as detailed in Section B.2.2. Efficient busy wait is also proposed there, using a lock-waiter state and a busy wait register.

*Feature 7: Flushing on Cache-to-Cache Transfer.* When transferring a block from one cache to another, there are two advantages to *flushing* it.

- If the block is *dirty: reliability* in the face of crashes is improved
- If the block may be *clean or dirty:* clean/dirty status need not be transferred

Keep in mind that a protocol supports cache-to-cache transfer only from a cache having source status for the block (indicated at the top of the table).

In view of this, if a source can have *either clean or dirty status* (Papamarcos and Patel, Katz et al., and the current proposal), then the clean/dirty status must be transferred along with the block, unless the block is flushed to memory while it is transferred — as it is in the Papamarcos and Patel scheme. Papamarcos and Patel, just the same, flush only dirty blocks, so clean/dirty status must, in effect, be put on the bus in their protocol, anyway. If memory can keep up with the flushes, as considered next, and if available bus codes are scarce, it may be useful to flush *all* blocks so that two different codes are not needed for cache-to-cache transfer.

However, the flush will slow down the transfer if the bus and memory do not support the concurrent flush at all, or at the speed of the caches. The cache speed can be *attained exactly* by an interleaved memory in which transfers always start at the beginning of a block. The speed can be *approached on average* by an interleaved memory in which transfers start at any module, or by any memory that is buffered adequately, though interleaved memory will require less buffering. Therefore, due to its advantages, flushing should be implemented if it can be done concurrently with the transfer at the speed of the caches. The non-flush option (NF,S) is depicted for the current proposal in order to explicate the option having the more complex protocol.

Frank does not have a source read-state; instead memory is the source for any read-shared block. So if a cache requests read-privilege of a block for which another cache has write privilege, the request cannot be serviced by cache-to-cache transfer. Rather

---

the cache from being accessed from the bus) throughout the operation. This is because it would create deadlock between two processors simultaneously read-modify-writing the same variable, or needing to access each other's caches during a read-modify-write of different variables.

the block must first be flushed to memory, returning source status to memory, and then fetched from memory. In extending Frank's protocol, Katz et al. added the source read-state in order to allow cache-to-cache transfer on this occasion.

*Feature 8: Number of Sources for Read-Privilege Block.* Under Papamarcos and Patel, if a block is in any cache, it is fetched from a cache, rather than from memory. However, there is a disadvantage here. For if the block has *read* status, several caches may have the block, so any such cache must arbitrate in order to select the actual source. This is done so that only one cache may interfere with its processor, and if necessary, for electrical reasons, to limit the number of devices driving the bus. However, this method of selecting a source *does slow down the cache-to-cache transfer* — increasing the bus traffic, as well as the processor wait.

Under the current proposal and that of Katz et al., on the other hand, arbitration of potential sources is never required. However, if a block has read status in several caches and the source purges the block (flushing it to memory if dirty), there will be *no source cache* for the block. So the next fetch of the block must be serviced by memory, which may be slower than fetching from another cache even if arbitration is required.

*If a fetch from memory is significantly slower than cache-to-cache transfer with arbitration,* then the expected cost of each of the two disadvantages, that of arbitration and that of fetching from memory, should be determined in order to decide which is expected to be worse. Let us illustrate with an intuitive evaluation of the *frequencies* (not the expected costs) of the two disadvantages as they relate to *instructions* — probably the primary source of read sharing. *All* cache-to-cache transfers of instruction blocks will be slowed down under Papamarcos and Patel. Whereas cache-to-cache transfer of instruction blocks will be slowed down under the current proposal *only on the next fetch after the source of a block purges the block (while the block still has a valid copy in another cache).* However, under the current proposal, the cache that most recently fetches a block becomes its source. This implements a *least-recently-used replacement algorithm across caches* if the cache that most recently fetches an instruction block will tend be the last to purge it. In this case, the disadvantage of the current proposal will be less frequent than that of Papamarcos and Patel.

Finally, notice that if the fetch from memory is significantly slower than cache-to-cache transfer with arbitration, then the fastest option for the current proposal is *to resort to arbitration when there is no source.* This will work in a system that does not support flushing dirty blocks on a cache-to-cache transfer (Feature 7, Option NF,S) Specifically, when a source cache is present at a block fetch request, it signals its presence on the bus by placing the clean/dirty/lock status of the block on the bus. Therefore, at a request for a block which has read privilege in some cache, every such cache will arbitrate only if it detects that there is no source cache. Just the same, if this situation is sufficiently infrequent, then the complexity of implementing the feature can be avoided, resorting, instead, to a fetch from memory.

*Feature 9: Writing without Fetch on Write Miss.* Under *write-without-fetch,* if the processor is going to write an entire block, the block need not be fetched on a miss, though the bus must be accessed in order to invalidate the block in other caches, as on a normal write. In order to implement it, the compiler must know when a processor block-write instruction will write an entire memory block. This may occur in initializing data and in saving process state at a process switch. In addition, the processor must have a way to inform the cache that the block-write will write an entire memory block.

*Feature 10: Purging Dirty Read-Privilege Block.* If blocks are not flushed in cache-to-cache transfer (Feature 7, Option F), a read privilege block can be dirty. When a source purges such a block, it must access the bus and flush the block to memory. But it is possible to arrange for another such cache to take source status and tell the purger to abort the flush, thereby reducing bus traffic. Just the same, this feature has a complication: the potential takers must arbitrate, or else several sources may result (giving rise to the arbitration of Feature 8). The time required for arbitration may offset any reduction in transfer time. However, if the bus can electrically support multiple drivers, then this one, probably infrequent, occasion is acceptable for avoiding arbitration and for allowing the resulting interference with the relevant caches. Yet if the frequency is low enough, the implementation expense will not be justified anyway. In short, the value of this option is highly questionable.

*Feature 11: Memory Mode.* The current proposal includes a memory-mode bus command. A *memory-mode* request is one that is ignored by all caches, and is implemented for reasons that include these: to allow system analysis — test, debug, measure; and to increase reliability — avoid reading an accidentally cached value for an address that should not be cached, such as the address of a memory-mapped register or port.

*Feature 12: Efficient Busy Wait.* The current proposal also implements highly-efficient busy wait, using a busy-wait register that waits for the unlock action to be broadcast, and which then interrupts its processor. This eliminates all unsuccessful retries from the bus, and allows a processor to work while waiting (Section B.2.2).

**Innovation Summary.** The evolution shown in Table 2 can be summarized by listing the innovations of each scheme, as shown in Table 3. Rudolph and Segall (1984), along with the Firefly and Dragon, are added from Section B.2.2.

**Feature Evaluation.** The extent to which any feature improves performance needs to be evaluated for the particular system of interest. This is accomplished by formulating a stochastic model of the access patterns of the processors for benchmarks, and then by evaluating the features through analysis and simulation, using the model. This will allow the frequency of each relevant event to be determined, and from this, the speed-up that each feature may offer in the system of interest. Papamarcos and Patel, and Archibald and Baer (1985), provide examples of this work.

**Table 3. Innovation Summary**

- **Classic (pre-1978)**
  - Identical dual directories
- **Goodman (1983)**
  - Cache-to-cache transfer (source state) for *dirty* blocks
  - Dual control units
  - Serialization of conflicting *single reads and writes* (not atomic read-modify-writes)
  - Fully-distributed read/write/dirty/source status
  - Flush on cache-to-cache transfer
- **Frank (1984)**
  - Bus invalidate signal
  - No flush on cache-to-cache transfer
- **Papamarcos, Patel (1984)**
  - Cache-to-cache transfer (source state) for *clean* blocks
  - Serialization of *atomic read-modify-writes,* but not allowing efficient busy wait
  - Fetch unshared data for write privilege — *dynamic* determination of unshared status
  - *Multiple sources* for read-shared block; a read-privilege source *arbitrates* before providing a block
- **Rudolph, Segall (1984) (Section B.2.2)**
  - *Efficient busy wait:* first write goes through to other caches
- **Firefly, Dragon (Section B.2.2)**
  - *Update-by-block (write-back):* for unshared data
  - *Update-by-word (write-through):* for shared data; allows efficient busy wait
- **Katz, Eggers, Wood, Perkins, Sheldon (1985)**
  - *Dirty* read state
  - *Dual-ported-read* directory and data-store
  - Fetch unshared data for write privilege — *static* determination of unshared status
  - *Single source* for read-shared (dirty) block — fetch from *memory* if source purges block
- **Current proposal (major features)**
  - Systematic terminology and conceptual development
  - Efficient locking and busy-waiting — *lock states, busy-wait register*
  - Consideration of interdirectory interference
  - *Single source* for read-shared block, but last fetcher becomes source, allowing *LRU replacement across caches*
  - Write without fetch on write miss

# C. Sleep-Wait and Service-Request Queuing Paradigm for High-Contention Atomic Operations

## C.1. Sleep-Wait and Service-Request Queuing



As for busy wait, in making sleep wait more efficient, it is of special interest to make it *faster*. With this bias in mind, we will see how enqueuing and dequeuing on a sleep-wait queue, or other priority queue, can be made very fast. Then we will look more specifically at the use of priority queues as service-request queues. And finally we will generalize the hardware-queue technique as a paradigm for implementing high-contention atomic operations in VLSI.

### *C.1.1. Usefulness of Hardware Sleep/Priority Queues*

**Sleep/Priority Queues — Usefulness.** The priority queue is a pervasive construct in an efficient multiprocessor system, for it is used to implement sleep wait, as well as to manage service requests. High priority processes and requests should be given attention first, so that the system will meet the performance demands of the users and algorithms, as well as manage the limited hardware resources efficiently.

Sleep wait can be elegantly implemented using P and V operations on semaphores, that is, queues with their accompanying counts. P and V will respectively decrement and increment the count, and may also enqueue or dequeue a process. Priority queues without counts are also a central tool in a multiprocessor system, allowing one process or processor to send service requests to another. For example, a program-interpretation (general-purpose) processor will send service requests to I/O processors, floating point processors, and other processors by enqueuing the requests on the appropriate queues.

The two pervasive uses of priority queues, then, are these:

- *P/V:* count decrement/increment is interlocked with enqueuing/dequeuing
- *Service requests:* service requests are sent from one process to another, typically from one processor to another

As we will see, the same hardware that is used for sleep-wait queues can be used for service-request queues, so two different kinds of queue need not be designed. Figures 21, 22 illustrate the P/V operations, which will be explained in detail below.

Also note that the sleep-wait operations that we will consider here are standard P and V, which do not implement multiple readers. Yet if *multiple-reader* sleep-wait were considered important, it could be implemented in hardware, extending the paradigm presented here for P and V. Or else the synchronization descriptor could be implemented in software, and the actual wait queues implemented using the hardware priority

queues. In this case, access to the synchronization descriptor should be achieved using busy wait; this is because contention for the synchronization descriptor should not be high, since the queue operations will be so fast. Finally, if it is desirable to implement *optional-wait* P operations in the hardware, this is possible as well.

**Hardware *vs.* Software Queues — Performance.** If priority queues are implemented in software, they will incur non-trivial performance cost in terms of processor time, as well as switch time, to fetch the needed data, and to write the data to memory if it cannot be left in a cache. If, in addition, the sleep-wait operations (P/V) are not implemented in hardware, access to any software queue will be possible only by busy waiting on the lock for the queue. *With hardware implementation of sleep/priority queues, on the other hand, busy wait can be eliminated and the switch time reduced to a trivial amount.* The amount of reduction in switch traffic, and the corresponding increase in performance, must be determined through stochastic modeling, informed by simulation, in order to be sure that the cost of the hardware is warranted by the performance gain in the system of interest.

*Elimination of Busy Wait.* Actually, the only way to reduce the probability of busy wait to zero, for a particular wait condition, is to allow the hardware sleep-wait queue to grow as long as the maximum number of processes that could ever wait on the condition. However, with queues for which this worst-case value is much larger than the average value, this would entail undesirable hardware cost due to a long, fixed-size queue, whose cells would be underutilized; or else it would entail a complex and slow algorithm to link the entries, or to extend the queue in some way when necessary.

The method developed here, instead, uses fast, fixed-size queues of reasonable size — determined by analysis and simulation — to reduce the probability of busy wait to an acceptably low value. If a queue fills up and a request is then made to place something on the queue, this rare occurrence is handled by the requester busy-waiting until the queue again has space available. While waiting, the requester can record the occurrence of the event in a table for future use in adjusting the size of the queue. Therefore, *elimination of busy wait* is a convenient way of saying that the probability is reduced to an acceptably low level, so that busy wait is rarely, if ever, seen.

*Parallel Switch.* It is interesting to notice that, while the most efficient implementation of busy wait requires a broadcast system — to allow the broadcast of current status information — the most efficient implementation of sleep wait allows a parallel switch without broadcast, since the information and operations are centralized, in a memory processor unit (MPU). The MPU performs the enqueuing and dequeuing operations on the queues (Figures 21,22). It is especially advantageous to have memory perform high-contention atomic operations, in contrast to other operations, since the requisite data cannot be kept in a processor cache for an extended period of time due to the contention for the data. The data must be fetched from memory or another cache anyway. The performance advantages will become evident in the ensuing discussion, and are summarized in Section C.2. In short, an attractive side-effect of having memory perform the

atomic operations is that this implementation is possible in a parallel switch.

*Ultracomputer Approach.* The designers of the Ultracomputer have recognized the need for fast queue operations (Gottlieb et al. 1983; Edler et al. 1985). However, their implementation is mostly in software, the only hardware atomic operation being *add.*

Atomic add is used to assign, to a requesting process, a queue cell for the process to insert into or delete from. However, *the assigning of cells is not interlocked with the enqueuing/dequeuing operations on the cells themselves.* The result of the lack of interlock is one complication after another, until the final scheme is incredibly complex. Complicated techniques are required for insuring that inserts and deletes on each cell are serialized properly, as well as for preventing overflow and underflow of the queue. For all of this effort, the result is nothing more than strict FIFO queues, requiring priority queues to be implemented by multiple FIFO queues, each covering a single priority value.

Furthermore, *busy wait is not eliminated;* P and V must still be implemented using busy wait. And busy wait itself is implemented *inefficiently* using atomic add instead of test-and-set. In particular, a *successful* busy-wait try requires *two* switch accesses, one to test the variable (to preclude livelock, where two opposing processes alternately increment and decrement the variable, neither gaining access), and one to atomically test-and-decrement it; while an *unsuccessful* try requires a *third* switch access to nullify the decrement by incrementing the variable. As they note, Dijkstra (1972) mentioned the possibility of using atomic increment for busy wait, but decided against it due to the livelock possibility, opting instead for a swap instruction, used to implement test-and-set. Gottlieb et al. argue that as the number of CPUs grows larger, the advantage of atomic add, here, increases. But it seems to me that atomic add has only disadvantages here: it generates twice as many switch accesses as test-and-set for a successful try, and three times as many switch accesses as test-and-set for an unsuccessful try.

These inefficient Ultracomputer algorithms appear to result from an attempt to use the atomic add operation to the fullest. Add is commutative and associative, so can be implemented by the Ultracomputer switch, as shown in Figure 23. Although the switch implementation of commutative-associative atomic operations is a truly creative contribution to computer architecture made by the Ultracomputer designers, you can go only so far with just an atomic add! For busy wait, atomic add is inefficient. And for efficient management of high-contention queues, especially sleep and priority queues, atomic add is of little help. Hardware enqueuing and dequeuing must be used, and since VLSI chip-design tools are now readily available, the time for hardware queues has arrived.

### *C.1.2. Structure, Management, Function of Hardware Queues*

**Data Structures.** There are two data structures that are used to implement sleep-wait queues

- *Hardware structure:* semaphore — containing queue and count
- *Software structure:* process control block — containing state of enqueued process

We will now look at each in turn.

*Semaphore.* A semaphore is an atom that contains a *count* and a *process queue,* and is used to manage access by processes to a resource pool using sleep wait. The term *semaphore* is also used to refer to the count itself, with the ambiguity resolved by context. If the maximum size of the pool is $n$, then the current size varies between 0 and $n$ as the items are allocated and deallocated. The count is initialized to the initial number of resource items available, usually $n$, and is subsequently operated on by increment and decrement. The resulting count indicates the following, where *size* refers to current size of the pool or queue.

- Count $\geq 0$ indicates
  - *Pool size* — number of resource items available
  - *Queue empty* — no processes waiting
- Count $\leq 0$ indicates
  - *Pool empty* — no resource items available
  - *Queue size* — number of processes waiting

A queue, in the present context, is a *priority queue,* providing deletion of the highest priority process in the queue. The low-level details of the design have not yet been developed, but the following is clear.

Each queue will consist of a header and a body. The header contains the current count and maximum possible count, while the body contains the queue cells, each of which has an ID field, a priority field, and a valid bit. The queue has a fixed number of cells, as mentioned earlier, for speed. The appropriate queue sizes, as well as the frequencies of each size, must be determined by analysis and simulation, but typical sizes may be 1, 2, 5, 10, 20, in decreasing frequency — there will be many small queues but only a few large queues. A single chip may contain queues of different sizes, or queues only of the same size. In the former case, all queues in a row on the chip should probably have the same size, in order to make the design more regular (Figure 24). The highest priority entry on a queue is determined in the way that bus arbitration is typically done — resolving each bit in turn, from the most significant to the least, among the valid cells. The queue could also be made *associative* — allowing deletion by process ID — if the cost appears warranted by the frequency of use.

The count is never allowed to increment above its maximum value. Consequently, a

queue that is not normally thought of as having a count, such as a ready queue or, more generally, a *service-request queue*, has the maximum initialized to zero, allowing the hardware to be the same for both kinds of queues.

The queue will also need logic for determining when it is full. One alternative is to use the valid bits — detecting that there are no invalid cells. Another alternative, is to maintain a hardwired size that would be compared to the current number of valid cells as indicated by a negative value of the current count.

In conclusion, the replication of a few different kinds of logic cells, in particular, count cells and entry cells, and the simple control, make an MPU queue chip highly regular, hence a good target for VLSI implementation.

*Process Control Block.* A process control block (PCB) is the data structure that contains the state of a process while the process is not running, and other necessary control information, if any. It has a *valid bit*, which is set to invalid when the process is awakened and its state is loaded into a processor, and is set to valid when its state is saved in the block as it goes to sleep. For service-request queues, similarly, a *service-request control block* would be used to pass the needed information between the requester and the server.

**Queue Management.** In order to insure the integrity of sleep-wait and service-request queues — whether they are implemented in software or hardware — the system software provides the user with the procedures for manipulating the queues — the procedures for initialization, enqueuing, and dequeuing. These procedures will collectively be called *the queue manager*, and will include the responsibilities of an operating system (OS) scheduler and dispatcher. In a high performance system, the overhead of invoking the queue manager will be as small as possible, close to that for any user-designed procedure, the only difference being invocation by trap instead of explicit control flow. Unlike a monitor, the code will be reentrant, allowing any number of processors to execute it simultaneously. More specifically, tables used to manage allocation of the queues will be lockable, but the code for manipulating the queues will not be lockable, thereby allowing greater concurrency. Consequently, at any one time there could be an instance of the manager running on each processor.

Since the hardware queues are of fixed size, the queue manager will allocate, address, and deallocate them as it would for fixed-size (statically-allocated) software queues. At the time of an allocation request, if there are no hardware queues available of the appropriate size, a software queue will be allocated instead. The information needed for subsequent access to the queue will be kept in a small descriptor. It would probably not be advantageous to make every queue a hardware queue even if that were possible. The main advantage is in implementing the *high-contention* queues in hardware.

Finally, as will be explained in detail in the section on operations, the queue manager does not directly manipulate a hardware queue. Instead, the MPU containing the queue

manipulates it. Or put differently, the MPU consists of all the queues, along with an interface between the switch and each queue. Each queue contains highly parallel circuitry that performs the enqueuing/dequeuing operations. The queue manager requests the init/enqueue/dequeue operation of the appropriate MPU, identified by the queue address, and in turn receives the information it needs on the current status of the queue.

To illustrate, in response to an enqueuing request, the manager is informed as to whether the queue is already full or not. The CPU running the queue manager then gives up the switch, having held it only for the time needed for the high-speed write/read — a write followed immediately by a read from the queue, through different registers in the queue. The addressed queue continues to carry on the requested operation, while at the same time the CPU evaluates the information it received in order to determine what to do. Notice the concurrency between the CPU and the MPU, and among the parallel components of the queue circuit, all achieved without holding onto the switch. Ideally the queue circuitry will have a cycle time as fast as the switch cycle time. But if current VLSI technology does not allow this, then successive requests to the same queue will suffer some delay.

**Queue Operations.** There are three operations that can be executed on a priority queue.

- *Initialize* the queue
- *Enqueue* an entry
- *Dequeue* the highest priority entry

Let us consider each in turn. When the queue manager is referred to, now, the reference is to the instance of the queue manager of interest.

*Initialization.* The queue manager initializes a sleep-wait queue by requesting its initialization, and by sending the initial and maximum values of the count along with the request. All cells of the queue are thereby set to invalid, and the current and maximum counts are set to the requested values.

*Enqueuing — P.* In requesting the P operation, the queue manager gives the request to the MPU, along with the address of the queue and the process's ID and priority (Figure 21). The MPU reads the count and returns the value to the manager, along with an indicator of whether the queue is full or not. This constitutes a write to queue registers, and a read from queue registers, and completes the switch transaction.

If the queue is not full, the MPU concurrently decrements the count; and if the resulting value is less than zero, enqueues the requesting process, placing the ID and priority on the queue. The hardware actually enqueues the process while operating on the count and evaluating it. If the queue was not full and the result is negative, then the valid bit of the new entry's cell is set to valid.

In the meantime, the queue manager decrements the count it received, and if the result

is negative and the queue was not full, it saves the state of the process in the process's PCB. It then takes a process from a ready queue and loads its state. The reason for the valid bit on the PCB is that it is possible, though highly unlikely, for a process ID to be moved from a wait queue to a ready queue and then to a CPU before the process state has been written to the PCB. If a CPU wants to run a process whose PCB is invalid, it busy waits on the valid bit. This should rarely, if ever, occur. On the other hand, if the queue was full, which also should be a rare occurrence, the manager busy waits until the queue is no longer full, as described earlier.

*Dequeuing — V.* The V operation is analogous to the P operation. In this case, the queue manager gives the MPU just the request and the address of the queue (Figure 22). The MPU reads the count and returns its value to the manager, as in the P operation, but now it also returns the ID and priority of the highest priority process on the queue (if any), effecting a read from queue registers.

In addition, the MPU concurrently increments the count, and if the initial value was negative, dequeues the highest priority process by setting the cell's valid bit to invalid. Note that when an MPU receives a V request, the highest priority process in each of its queues should already available, having been selected by the queue circuit during the time since the last insertion or deletion on the queue. As mentioned, this selection should be as fast as the switch cycle, if possible; otherwise successive requests to the same queue will suffer some delay.

In the meantime, the manager evaluates the count it received, and if it is negative, puts the process ID and priority that it received from the queue on a ready queue, after adjusting the priority appropriately. The priority is an increasing function of the amount of contention expected on the lock, but it may be as simple as giving any lock holder the same high priority.

In addition, it is a software error for the number of V's on a semaphore to outbalance the number of P's so as to (attempt to) push the count above its maximum. This error can be returned to the queue manager in response to a V request, if desired. Since the initial value of the count is kept in the queue header and the current value is never allowed to exceed it, the same circuit that manages this can return the above error signal. If the queue, on the other hand, is used as a service-request queue instead of a sleep-wait queue, the error line would be ignored by the queue manager when requesting a dequeue operation.

*Priority Preemption.* Priority preemption of a running process may be desirable. Specifically, if a running process has lower priority than the process just taken from a sleep-wait queue, it may be desirable to preempt the running process. With a non-broadcast parallel switch, preemption is limited to the process invoking the V operation on the wait queue. The queue manager should compare this priority to that of the process taken from the sleep queue. If the priority of the running process is lower, the manager should save the state of the running process, place the process on a ready

queue, and load the state of the process taken from the sleep queue.

Under a full broadcast system, on the other hand, broadcast can be used to achieve *priority preemption across all processors whenever a process is put on a ready queue.* Specifically, each CPU is given a *priority register* containing the priority of its currently running process, along with hardware that will compare this value to a broadcast priority, will arbitrate for the bus if the broadcast priority is higher, and will interrupt the process running on the CPU if the arbitration is won. As mentioned in Section B.1.1, this hardware may be integrated with a CPU cache.

At a V operation, when the queue manager receives a process priority and ID from an MPU, it places the priority and ID on a ready queue, and in doing so, broadcasts the priority to all CPUs. The broadcast priority is, accordingly, compared to the priorities in all of the priority registers. So if a running process has a lower priority and has the corresponding interrupt enabled, its priority-register interface arbitrates for the next bus system, as in efficient busy wait. The arbitration scheme, as discussed in Section B.2.2, uses the most significant bus-priority bit. The bus priority, in the present case, is determined by the setting of the most significant bit, along with the boolean complement of the process priority (or a derivative of it). The complement is used because the lower the process priority, the higher the bus priority should be, in this case. There must be a way to resolve equal process priorities, say using a daisy chain, or having the lowest bits of the bus priority be determined by the processor.

The winner then interrupts its CPU, and the queue manager there saves the state of the running process, places the process on a ready queue, and loads a new process from the ready queue that occurred in the corresponding broadcast. If the system has a single bus, then there is probably just one global ready queue (Figure 25).

**Service-Request Queues.** As mentioned, priority queues have frequent use as request queues for processes or processors. To illustrate, let us consider how they may be used for an I/O service request motivated by an I/O request or a page fault, either of which invokes the appropriate OS procedure/s.

The OS forms an I/O request by creating an *I/O control block*, comparable to a process control block, and enters the information that the I/O driver will need to service the request. The OS also includes the process ID, priority, and (if appropriate) the processor ready-queue address in the I/O control block. The OS then gives the I/O request an ID, as well as a priority, and enters the ID and priority in the appropriate I/O request queue.

An I/O driver takes requests from the appropriate queue, mapping to the corresponding control blocks in the same way as done for process control blocks, and services the requests, leaving the resulting status information in the I/O control blocks. When done with a request, the driver puts the process ID and priority on the appropriate ready queue.

### *C.1.3. Related Issues*

Let us now look at several issues that have yet to be resolved. A few of these have already been mentioned.

**Fast Address Mapping.** How should fast mapping from a process ID to the address of the process control block be implemented? If processors have their own ready queues, how should fast mapping from a processor ID to the address of the processor ready queue be done? The best way may be to take the process ID as a lower part of the PCB virtual address, and to take the processor ID from a lower part of the processor ready queue virtual address. The upper part of the addresses will then be fixed — devoting a certain area of the address space to PCBs, another to processor ready queues, and other areas to other kinds of shared objects. Note also that if the caches are managed by virtual address, rather than physical, the synonym problem can be resolved by devoting the upper part of the address space to shared data, and this resolution is consistent with the fast mapping scheme.

**FIFO Queues.** It may be desirable to have fast FIFO queues as well as priority queues. These can be implemented with the priority queue hardware by allowing it to be initialized to a FIFO mode, which will operate as follows.

The enqueuing operation will remain the same — the first empty cell is selected — but the dequeuing operation will be changed so as to choose the oldest entry. It would be possible to have the dequeuing operation cause all entries to move one cell toward the head of the queue, as in a normal hardware FIFO. However, since the priority hardware is in place, it will be less expensive to *use a single priority bit, and have this move from cell to cell.*

To be specific, suppose that the value one is of higher priority than zero, and that all entries have the most significant priority bit zero, except for the oldest entry. Then the normal dequeuing operation will select the oldest entry. In addition, the dequeuing operation should cause this one to move to the next cell in the direction of the queue tail (wrapping around to the other end of the queue when necessary), which will be the new oldest cell. When an entry is loaded, it is given a most significant bit of zero, unless the queue is empty (the semaphore count is non-negative), in which case it receives a most significant bit of one.

**Starvation.** Should starvation of a process be prevented in the case that there is always another process of the same priority on the queue? What is the probability of starvation without its prevention?

In case it is decided that starvation should be prevented, the following scheme may be the simplest. Let a fourth command to a queue be implemented:

- *Promote:* promote, or raise, the priorities of old entries to at least the value sent with the command

In 4.2BSD Unix, the OS evaluates the priorities of processes in the ready queue every second and raises the priorities of those that have been in the queue a relatively long time (Peterson, Silberschatz 1985). So let us assume that every so often the queue manager will execute the promote command on the queues of interest. Further, let us assume that only the processes having priority less than $v$ are of concern. Then the queue manager will give the value $v$ with the command, and the queue circuitry will then modify the priorities of the *old* processes so that they become at least as great as $v$.

Regarding the age of the process, the simplest implementation is to not maintain an age indicator and just apply the command to all entries in the queue. A simple age indicator, on the other hand, is a single *age bit* for each cell, where zero means *young* and one means *old*. Under this scheme, when an entry is placed into a cell, its age bit is cleared to zero. When the promote command is subsequently given, it is applied to all entries whose age bit is one, and then all age bits are set to one.[16]

Regarding raising the priorities, let us assume that larger numbers designate higher priorities. The argument $v$ can then be bitwise OR'd with the priorities of the old processes. This can be implemented indirectly by making the priority bits of a cell individually loadable, and having the bits of $v$ that are one trigger the *loading* of the corresponding priority bits of the old entries. Alternatively, the ones of $v$ can trigger the *setting* (instead of loading) of those bits if that is simpler. The result is the same, in either case — the bits take on the value one. (If smaller numbers designate higher priorities, then $v$ is bitwise AND'd, which is equivalent to the zeros of $v$ triggering loading or clearing.)

To be specific, let us assume that a priority has seven bits and that $v$ is 100 (binary). Then an entry having priority $xxxx0xx$ would be given the new priority $xxxx1xx$ (where the $x$'s are not necessarily equal). Notice that this not only raises priorities $00000xx$, but also raises priorities that are greater than $00001xx$ if their bit two is zero, thereby erasing some distinctions among higher priorities. This is a side effect of a simple hardware algorithm. However, it should not cause a problem since the effects of priority distinctions cannot be finely tuned anyway because of all the other variability in a multiprocessor computer system. Note also that the PCB of a process contains other information about the priority of a process, so the change to the priority in a queue will not destroy this PCB information.

**Associativity.** Will associativity be worth implementing? When the software wants to kill a particular process that is on a queue, will it know what queue to look on? If not, should it search several queues, or else (in a broadcast system) enter the process ID in a global, associative table, to be accessed whenever a process is enqueued or dequeued?

[16] The most general scheme — of theoretical interest only — is to have an age counter for each cell. When an entry is enqueued, its counter is set to a high value of interest, say all ones. At each promote command, the command is applied to all entries having value zero, and all counters not yet zero are then decremented.

**Switch Interface.** Only three commands are needed for the queue operations under the original scenario, one command for each of the operations init, enqueue, dequeue. On a standard bus, these commands would be implemented using memory *write* for init and enqueue, and memory *read* for dequeue, along with one other bit to distinguish between init and enqueue. If the bus did not have a user-specifiable bit, the second bit would be one of the data bits.

If FIFO mode is implemented, another init command will be needed. If starvation is to be precluded using the promote command, a fifth command is needed. And if deletion by process ID is desired, a second dequeuing command must be included, bringing the total to six.

**Partitioning.** If the collection of processes and process queues in the system can be partitioned so that processes in a set primarily access queues in that set, then local MPUs may pay off (Figure 26). In this case, a set will be assigned to one processor and its MPU so that processes running on that processor need not access the switch in order to access the queues in their set. Such partitioning and assignment to processors will also improve the effect of the caches as local memories, since the processes running on a processor will now tend to access the same shared data, rather than data accessed by processes on other processors. Under this organization, preemption by higher priority processes would be undesirable across processors and thus would be confined to each processor, as with a non-broadcast parallel switch.

**Entry Information.** What information should be placed in a queue entry? May it be desirable to enqueue information other than just the process ID and priority? For example, under partitioning, a processor ID should be included, indicating the processor that the process should be run on: when taken from a sleep-wait queue, the process would be put on that processor's ready queue.

## C.2. Paradigm for High-Contention Atomic Operations

A general paradigm has been illustrated here using sleep/priority queues:

*Implement high-contention atomic operations in VLSI circuits at the memory cells.*

We may call a chip containing these circuits an *MPU* since it may be thought of as an intelligent memory unit.

The advantages of this paradigm are less switch traffic and greater concurrency. Briefly, the CPU is relieved of executing the atomic operations, the bus is relieved of carrying the corresponding traffic, and the MPU executes the atomic operations themselves at high speed. More specifically, we have the following.

- Less switch traffic
  - Atom is not transferred to CPU, but is left at MPU
  - Little data is transferred between CPU and MPU
  - Switch is not held during CPU operation
  - Atom is not locked or busy-waited on
- Greater concurrency
  - Concurrency between CPU and MPU
  - Concurrency in MPU circuit
  - Concurrency among processes accessing atom
  - Concurrency among processors accessing the switch

Let us look at each advantage, in turn, then at two final considerations — the idea of a general purpose MPU, and the suitability for VLSI implementation.

**Less Switch Traffic.** Use of the memory switch is conserved by leaving the atom at the MPU — where the MPU executes the atomic operation — rather than transferring it from CPU to CPU. If the atom is small, in particular, one bus data-word, there is no significant saving here. But if parts of it may occur on several memory blocks, then it is a significant help to avoid the frequent switch accesses otherwise necessary to transfer all of it from CPU to CPU. Keep in mind that since the operation of interest is a high contention operation, even if there are caches, the atom will probably not stay in a cache very long after being used there, but will soon be fetched by another cache for use by its CPU. Consequently, caches offer little, if any, help in reducing the frequency of transfer from CPU to CPU.

The CPU could alternatively leave the atom in the MPU but hold the switch while it operates on it, fetching just the words of interest. Or it could busy or sleep wait for access, eventually locking the atom, then fetch just the words of interest, and finally unlock the atom. But any such scheme consumes more bus cycles than the MPU scheme.

The MPU alternative is to transfer a small amount of data from the CPU to the MPU, along with the request and address, and then to transfer a small amount of data from the

MPU back to the CPU, including current status information. To illustrate, for a P operation, a process ID and priority are sent to the MPU, and the current count value and full indicator are read out and sent to the CPU. This exchange would ideally take place as fast as a read from high-speed memory: the first phase is the address phase, during which the address, request, and data are latched by the addressed queue; while the second phase is the data phase, during which the data is read from the queue registers and latched by the CPU. However, if the same data lines must be used in both phases, then the transfer will probably require two switch cycles, rather than just one.

**Concurrency.** After the CPU receives the status information, it no longer needs the switch, so releases it. The CPU then evaluates the status information, say the count value and the full indicator, and the MPU continues to execute the operation it started upon receiving the request. The concurrency between the CPU and MPU, here, is what allows the CPU to release the switch so soon, not holding it while it or the MPU executes the operation.

In executing the request, furthermore, the MPU utilizes the parallel logic in its circuit to execute the operation faster than software could ever execute it. Hopefully the circuit is fast enough to service successive requests as fast as could be delivered on the switch. But if the MOS technology of the day does not meet this speed, successive requests to the *same atom* will suffer some delay, though successive requests to different atoms in the same MPU will incur no delay.

Notice, furthermore, that since no locking and unlocking of the atom is necessary, the atom is available whenever the switch connected to it is available (excepting delay that may be unavoidable in successive requests to the same atom). And since the operation executed by the parallel circuit is much faster than a software implementation — as fast as a read from high-speed memory — there will be less contention for access to the atom. Consequently, there is greater concurrency among the processes in pursuing their activities since they are not waiting as long to gain access to the high-contention atoms.

Finally, implicit in all of this is the implication that since there is less switch traffic, there is greater concurrency among processors in the system. Furthermore, broadcast is not needed for this algorithm, as it is for high-speed synchronization of caches, so this algorithm can easily be implemented in a parallel switch.

**General Purpose MPU.** One may ask, why not use a general purpose processor to manage the atomic operations at memory, thereby allowing many different kinds of operations to be done? Queues, for example, could then be variable size. The reason is that the generality would be purchased at the expense of speed, and speed is essential because the atomic operations of interest are high contention. An MPU will only pay for itself in dealing with high-contention atomic operations; low contention operations can conveniently be done by the CPUs.

**Suitability for VLSI Implementation.** Finally, the kind of MPU envisioned is

appropriate for VLSI implementation since it will consist of a highly regular structure. To be specific, a few kinds of cells, say queue-entry and queue-header cells, will be replicated many times on a chip, and will be managed by relatively simple control.

One hears over and over that computer designers must have a vision for what technology offers them, so that they can capitalize on it. This is my vision.[17]

[17] Designing one of these chips appears to me to be no more complex than a project for the introductory VLSI course at Berkeley. Sophisticated design experience can be important, just the same, in making the circuit cells as compact and as fast as possible. In fact, these two issues are the major low-level questions still in my mind regarding queue circuits: How many queues of what size can fit on a chip? How fast can a queue operate?

# D. Conclusion

# D. Conclusion



## D.1. Vision for Fast Synchronization

We have explored the three low-level synchronization issues for shared-memory architecture and proposed new mechanics for corresponding high-performance implementation.

- Synchronization of caches — for broadcast switch
- Implementation of busy wait — for broadcast switch
- Implementation of sleep wait — for any switch

Let us now recall the two reasons for using busy wait.

- A situation where busy wait is *less costly* than sleep wait
- A system where busy wait is *necessary* in order to implement sleep wait

If the hardware implementation of sleep-wait queues proves itself — if the performance advantage in the system of interest is worth the additional hardware cost — then the second occasion for busy wait will vanish. In such a system I envision sleep wait costing very little in bus traffic or processor time. But in any other system, the second reason for busy wait remains, making *efficient busy wait* critical:

- Eliminating unsuccessful retries from the bus
- Relieving a waiting processor of polling the status of a lock, thereby allowing it to work while waiting

The efficient busy-wait scheme developed here fulfills both of these requirements, and in addition, generates no interference with any processor's use of its cache.

Furthermore, where synchronization does not involve one process waiting for another (as it does in a producer/consumer relationship), I envision *fast synchronization operations based on busy wait, in which waiting is rarely necessary.* I imagine the processes in the system communicating rapidly using fast operations whose target atoms are each contained entirely on a single memory block. The efficient locking scheme introduced here will enable a cache to fetch an atom for write privilege and lock the block as it arrives and the first operand is read. The processor will then execute a fast operation on the atom and unlock it with the final write to the block. The locking, the operation, and the unlocking are fast and efficient. In particular, *the locking and unlocking incur no processor time or bus traffic in this case.*

In short, my vision for high-speed synchronization is that it may prove strategic for *hardware to implement sleep-wait queues,* thereby realizing high-speed sleep wait with minimal bus traffic and processor time. But if this idea does not prove itself, I look to *highly-efficient busy wait,* which incurs no unnecessary bus accesses and allows a

processor to work while waiting. Finally, I envision *fast synchronization operations on small atoms*, where locking and unlocking incur no processor time or bus traffic at all.

## D.2. Best of Both Worlds

We have seen three examples of the ability of broadcast to synchronize or coordinate a group of devices at high speed.

- Synchronize *caches* — read/write sharing of data
- Synchronize *processors* — notification of lock release in busy waiting
- Synchronize *processors* — preemption of process by higher priority process

Broadcast is possible with both parallel switch and single-bus; however it is clearly more expensive with a parallel switch, requiring $n$ buses.

Following the innovative idea of Professor Al Despain, in Project Aquarius at Berkeley we propose to incorporate into the same system *the best of both worlds — that of the parallel switch and that of the bus.* (Dobry, Despain, Patt 1985) The parallel switch allows many processors to access main memory concurrently, while the bus provides high-speed synchronization operations at reasonable cost. In particular, we recall that the hardware needs to serialize conflicting access requests only for hard atoms, since the software serializes access to soft atoms. Therefore, we propose *an architecture having two switch-memory systems:* one system for hard atoms and software synchronization structures, using a bus; and one system for all other objects, using a parallel switch.(Figures 27,28)

- Synchronization system — bus
  - *Hard atoms:* counters, locks, sleep/priority queues, and such
  - *Software synchronization structures:* reader/writer synchronization descriptor/queues, and such
- Data system — parallel switch
  - *Instructions*
  - *Unshared data*
  - *Soft atoms* (other than synchronization structures): shared buffers, and such

The major activity of the *synchronization system* will be to provide high-speed synchronization among the processors, and among processes running on the same processor. The operations in the synchronization system will include fast operations on small, hard atoms, as well as more complex operations on structures such as reader/writer descriptors/queues. These operations will use the lock state in the caches, if there are caches, or else will use a single lock register, to implement busy-wait locking on the atoms. Efficient waiting using a busy-wait register will also be employed. In addition, the synchronization system will implement sleep and service-request priority queues. If the performance benefit of the hardware priority queues appears warranted, they will be implemented in VLSI to provide fast queuing operations and to eliminate busy wait.

It is not yet clear if caches will pay off since most, if not all, operations in the synchronization system will be read-modify-writes on high-contention atoms. In this case, the hit rate on reads will be close to zero, but it will be close to one for writes, and nearly all blocks in a cache will be be dirty. Therefore, *the purpose of the caches, as opposed to faster processor-registers, would be to reduce the number of writes on the bus.* This would occur if a cache never had to purge a (dirty) block when fetching a new one — because the block that was previously in the target frame had already been fetched by another cache. So if no purging is necessary, the caches would reduce the data read/write traffic on the synchronization system bus by half by eliminating the writes, in the read-modify-write operations. The rest of the bus traffic would consist of unsuccessful busy-wait tries and hardware queue accesses.

In contrast, the *data system* will contain most soft atoms, unshared writable data, and read-only data. The major activity of this system will be to access and to share large data structures — sharing that will be synchronized in the synchronization system — as well to access unshared and read-only data. This system will also maintain the state of the processes in their process control blocks, since a process state will be too large to manage in the synchronization system, which will be devoted to relatively fast operations on small variables and structures. The cache synchronization system, here, will be a non-broadcast system, as described in Appendix 4.[18]

### D.3. Evaluation of Features

The next step is to evaluate the proposed architectural features for fast synchronization in the context of a particular system of interest in order to determine how beneficial each feature is in that system — to determine if each feature is worth its cost in that system. *A stochastic model informed by simulation* will enable this evaluation to be made. In Project Aquarius, our focus will be on evaluating these features with respect to the multiprocessor architecture shown in Figure 28, which is being designed as a high performance system to execute Prolog. I am currently formulating a stochastic model to use in making the evaluation.

[18] A precedent for the split-level, tightly-coupled Aquarius architecture is the loosely-coupled, dataflow architecture of Srini (1980; 1985). The latter has two communication systems, one for urgent, short communications, and another for normal-priority and long communications.

## References


Archibald, J., Baer, J-L. "An economical solution to the cache coherence problem." *11th Ann. Intl. Symp. Comp. Arch.*, **1984,** 355-362.

Archibald, J., Baer, J-L. "An evaluation of cache coherence solutions in shared-bus multiprocessors." U. of Wash. Tech. Report 85-10-05, October **1985.**

Bell, C.G., Burkhart, H.B. III, Emmerich, S., Anzelmo, A., Moore, R., Schanin, D., Nassi, I., Rupp, C. "The Encore Continuum." *AFIPS Conf. Proc., NCC,* Vol. 54, **1985,** 147-155.

Borriello, G., Eggers, S., Katz, R., McKinley, H., Perkins, C., Scott, W., Sheldon, R., Whalen, S., Wood, D. "Design and implementation of an integrated snooping data cache." Tech. Report UCB/CSD 84/199, U. of Calif. Berkeley, Jan. **1985.** The sequel to this report is Katz et al. (1985).

Censier, L.M., Feautrier, P. "A new solution to coherence problems in multicache systems." *IEEE Trans. Comput.*, C-27(12), Dec. **1978,** 1112-1118.

Colwell, R.P., Hitchcock, C.Y. III, Jensen, E.D., Sprunt, H.M.B., Kollar, C.P. "Computers, complexity, and controversy." *Computer,* 18(9), Sept. **1985,** 8-19. Rebuttal: Patterson, D., Hennessy, J., "Response to 'Computers, complexity, and controversy,'" *Computer,* 18(11), Nov. 1985, 142-143.

Denning, P.J. "Virtual memory," *Computing Surveys,* 2(3), Sept. **1970,** 153-189.

Denning, P.J., Dennis, T.D., Brumfield, J.A. "Low contention semaphores and ready lists," *CACM,* 24(10), Oct. **1981,** 687-699.

Dijkstra, E.W. "Hierarchical ordering of sequential processes." In Hoare and Perrott (ed.s), *Operating Systems Techniques,* NY: Academic, **1972,** 72-93.

Dobry, T.P., Despain, A.M., Patt, Y.N. "Performance studies of a Prolog machine architecture." *12th Ann. Intl. Symp. Comp. Arch.*, **1985.**

Edler, J., Gottlieb, A., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., Snir, M., Teller, P.J., Wilson, J. "Issues related to MIMD shared-memory computers." *12th Ann. Intl. Symp. Comp. Arch.*, **1985,** 126-135.

Frank, S. "Tightly coupled multiprocessor system speeds memory-access times." *Electronics* , Jan. 12, **1984.** This article describes the Synapse computer system.

Gajski, D.D., Peir, J-K. "Essential issues in multiprocessor systems." *Computer,* 18(6), June **1985,** 9-27.

Goodman, J.R. "Using cache memory to reduce processor-memory traffic." *10th Ann. Intl. Symp. Comp. Arch.*, **1983,** 124-131. An updated version is U. of Wisconsin Computer Sciences Technical Report #580.

Gottlieb, A., Lubachevsky, J., Rudolph, L. "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processes." *ACM Trans. Prog. Lang. and Sys.*, 5(2), April **1983,** 164-189.

*Guide to Parallel Programming.* Portland, OR: Sequent Computer Systems. June 3, **1985.**

Hill, M.D., Smith, A.J. "Experimental evaluation of on-chip microprocessor cache memories." *11th Ann. Intl. Symp. Comp. Arch.*, **1984,** 158-166.

Katz, R.H., Eggers, S.J., Wood, D.A., Perkins, C.L., Sheldon, R.G. "Implementing a cache consistency protocol." *12th Ann. Intl. Symp. Comp. Arch.*, **1985,** 276-283. This is the sequel to Borriello et al. (1985).

Kepecs, J. "Lightweight processes for Unix." *Usenix Conf. Proc.*, June **1985,** 299-308.

*MC68000 16-Bit Microprocessor User's Manual.* Englewood Cliffs, NJ: Prentice-Hall, **1982.**

Norton, R.L., Abraham, J.A. "Using write back cache to improve performance of multiuser multiprocessors." *Intl. Conf. on Par. Proc.*, **1982,** 326-331.

Papamarcos, M.S., Patel, J.H. "A low-overhead coherence solution for multiprocessors with private cache memories." *11th Ann. Intl. Symp. Comp. Arch.*, **1984,** 348-354.

Patterson, D.A. "Reduced Instruction Set Computers," *CACM,* 28(1), Jan. **1985,** 8-21.

Peterson, J.L., Silberschatz, A. *Operating System Concepts.* Reading, Mass: Addison-Wesley, **1985.**

Rector, R., Alexy, G. *The 8086 Book.* Berkeley, CA: Osborne/McGraw-Hill, **1980.**

Reed, D.P., Kanodia, R.K. "Synchronization with eventcounts and sequencers." *CACM,* 22(2), Feb. **1979,** 115-123.

Rudolph, L., Segall, Z. "Dynamic decentralized cache schemes for MIMD parallel processors." *11th Ann. Intl. Symp. Comp. Arch.*, **1984,** 340-347.

Smith. A.J. "Characterizing the storage process and its effect on the update of main memory by write through." *JACM,* 26(1), Jan. **1979,** 6-27.

Smith, A.J. "Cache memories." *Computing Surveys,* 14(3), Sept. **1982,** 473-530. Smith (1984) is an updated version.

Smith, A.J. "CPU Cache memories." Draft April 24, **1984.** To appear in M. Flynn and G. Rossman (eds.), *Handbook for Computer Designers.* This is an updated version of Smith (1982).

Smith, A.J. "Cache evaluation and the impact of workload choice." *12th Ann. Intl. Symp. Comp. Arch.,* **1985,** 64-73.

Srini, V.P. "An extended abstract dataflow methodology for designing and modeling reconfigurable systems." Ph.D. dissertation, U. of Southwestern Louisiana at Lafayette, July **1980.**

Srini, V.P. "A fault-tolerant dataflow system." *Computer,* 18(3), March **1985,** 54-68.

*System Technical Summary.* Portland, OR: Sequent Computer Systems. **1984.**

Wilkes, M.V. "Slave memories and dynamic storage allocation," *IEEE Trans. EC-14,* April **1965,** 270-271.

**Acknowledgements**

I thank Peter Denning for providing me the opportunity to pursue this research at RIACS, summer 1985.

I am deeply grateful to my advisor Professor Alvin Despain for the many hours that he has given toward discussing issues considered here, and for making many valuable suggestions, a few of which are explicitly acknowledged in the text. As I was writing this report, Al and I took time out to distill the essence of the ideas on caches and busy wait, and to submit a paper to the 1986 International Symposium on Computer Architecture. I used this distillation, in turn, in continuing my work on this report.

I thank Vason Srini and Steve Melvin for generously offering their time to carefully read the first version of this report and provide extensive criticism. I thank George Adams who, along with Steve, helped me gain valuable perspective concerning Al Despain's idea of full broadcast in a parallel switch.

I also thank Peter Denning, Luis Felipe Cabrera, Jung-Herng Chang, and Chien Chen for their comments. I appreciate valuable interactions with Professors Yale Patt, Jim Goodman, David Patterson, John Ousterhout, and Alan Smith, as well as with Matt Bishop, Bob Brown, Tep Dobry, Barry Fagin, Mike Karels, Gene Levin, Rick McGeer, Mike Shebanow, and John Swensen. And I am grateful to the other members of RIACS and Aquarius, and to the Computer Science Division staff, for their ongoing support.

This work was supported by RIACS grant NAS 2-11530, and DARPA (DoD) order 4871, monitored by Naval Electronics Systems Command under contract N00039-84-C-0089.

Finally, I want to express my affectionate appreciation to a very special friend. She has supported me through many long nights of work: my hamster Spunky. She worked along with me, scampering around my room chewing everything that was chewable, as well as most things that weren't. If the corners are missing on some of your pages, as they are on mine, this explains why.

# Appendices

## Appendix 1

## High-Speed Memory Transfer

As stated in Section A.2, Professor John Ousterhout has pointed out that computer architects can help operating system (OS) designers greatly by devising a fast data transfer operation, allowing fast I/O and fast memory-to-memory transfer.

My solutions to this problem are two.

- *Direct Memory Transfer (DMT):* Direct memory-to-memory transfer is managed by the memory units themselves. Transfer occurs at the maximum possible speed — the clock rate.
- *Cache-Mediated Transfer (CMT):* A processor cache mediates a memory-to-memory transfer using two DMT transfers. Hence the overall speed is half that of DMT, but entails much simpler hardware and software.

Let us look at each in turn.

**Direct Memory Transfer (DMT).** Three principles will enable architects to realize the full speed potential of memory and switch transfer.

- *Interleave:* Memory access must be pipelined using interleaving. The switch must support it, allowing split address and data phases for reading.
- *Direct transfer:* The two memory units involved in a transfer (main memory units, I/O processor buffers, or caches) must manage the transfers themselves so that the data moves directly from one unit to the other, crossing only the switch and necessary drivers and latches. No third processor should get in the way.
- *Switch synchronization:* The switch allows data transfers to run as fast as the clock, no handshaking necessary.

In short, *one memory unit directly transfers to another as fast as the clock,* hence an appropriate name is *direct memory transfer (DMT).* This is similar to direct memory access (DMA), but in the present case a main memory unit (not just a cache or I/O processor) has a control unit that allows it to initiate a transfer.

The reason that a main memory unit needs this power is so that high speed transfers can be made from one part of main memory to another. But in order for this to be possible, main memory must be split into at least two units, each interleaved and controlled as stated. This requires the understanding of the OS. The OS needs to be aware of the address bounds of the memory units so that it can plan high-speed transfers from one unit to another. Under a virtual memory system, this can be implemented by reserving several pages in each unit for buffers to be used for this purpose.

It may not be convenient for the OS to arrange this, and portability of the high-speed transfer may require some work. Consequently, let us explore a second memory-to-memory transfer method that is about half the speed of the foregoing scheme, but avoids

the hardware and software complications noted.

**Cache-Mediated Transfer (CMT).** As mentioned in Section B.1.1, a CPU cache can be used to implement fast memory-to-memory transfer. It simply needs to understand a CPU command to fetch a block from one address and write it to another. The cache will not bother storing the block in its data cells; it simply reads the block into its bus assembly register and then writes it from there. (Smith (1982) and Borriello (1985) illustrate the internal components of a cache, for the interested reader.)

In order to synchronize with the other caches, the cache should fetch from the first address with read privilege, then get write privilege to the second address (invalidating it in its own directory, if present) as it finally writes to the second address, as in an input operation (Section B.2.2).

The speed of cache-mediated transfer is about half that of direct-memory transfer because two DMT transfers are necessary per block. In addition, the CPU must command the cache to transfer block by block, unless the cache is given the intelligence to transfer all blocks within a particular address range.

The advantage of CMT, though, is much simpler hardware and OS software. The cache is already in place; its control just needs to be a little smarter. And the OS will not need to be concerned about the bounds of memory units.

**Intra-Memory-Unit Transfer.** Finally, one could give a memory unit the capability to do what the cache does, thereby freeing the switch. A memory unit could read out a block into its data registers, then write the block back to itself. The target addresses should already be available in its registers.

## Appendix 2

### Processor Requests, Cache Responses, Bus Commands

Details of processor requests, and the resulting cache responses and bus commands, are given in the table that follows. In order to avoid cluttering the table, the following events are omitted.

- *Hit:* If a cache has a block that is requested for read privilege via the bus, the cache indicates this by signaling *hit* on the bus, so that the requester will not assume write privilege if there is no source cache.
- *Source:* If a source cache is present, it provides the block, along with its clean/dirty/lock status, in response to a fetch request. The provision of the status indicates the source's presence to main memory, which must then refrain from driving the bus. Also, the cache that fetches the block becomes its new source.
- *Clean/Dirty:* When a processor writes a block, the block becomes dirty.
- *Lock Waiter / Busy Wait:* When a locked block is requested by another cache, the status becomes *lock-waiter* in the cache holding the block, and the requester's *busy-wait register* is loaded.

Several examples will be considered now to show how the table is read. Keep in mind that the figures will not be referred to here, but they do elucidate many of the details.

Suppose that a processor makes a request to read a word from its cache. The first four lines of the table show the sequences of *cache responses* that may be generated by this processor request. The four major columns present the four major cases: the current status of the block in the *processor's cache* is invalid (or absent), read, write, or lock. Suppose that the status is invalid or absent (a miss). Then the first line indicates that the processor's cache presents a request to the bus to gain read (R) privilege to the block and to fetch (F) the block. The next three lines present four minor columns showing the four alternatives: the status of the block in *another cache* is invalid (or absent), read, write, or lock (I/R/W/L). If the status is invalid or read, it remains invalid or read; if write, it changes to read; and if lock, it stays lock (I/R/R/L). The resulting status in the requester cache is, respectively, write (if invalid in *all* other caches), read, read, and invalid (or absent) (W/R/R/I). The cache, in turn, provides the target word to its processor, or garbage if the block was locked. (If hint read is implemented, the cache will have the actual word for its processor.)

To consider another example, suppose the processor requests its cache to lock a block, and the status of the block in the requester cache is read. Then the cache presents a request to the bus to gain write (W) privilege to the block, but does not present a request to fetch the block (no F), since it already has an up-to-date copy of it. If the block is present in another cache, it can have only read status, since it has read status in the requester cache. The status changes to invalid in that cache, and it changes to lock in the requester cache. Finally, the requester cache provides the target word to its processor.

**Table. Details on Processor Requests, Cache Responses, and Bus Commands[0]**

| *Processor Request to Cache* | *Sequence of Cache Actions* | Bus Commands and Cache Responses: *Initial Status in Requester Cache* — Invalid (or Absent) | | | | Read | Write | Lock |
|---|---|---|---|---|---|---|---|---|
| **Read** | *Requester cache to bus* | R, F | | | | — | — | — |
| | *Other cache initial status* | I | R | W | L[1] | — | — | — |
| | *Other cache final status* | I | R | R | L | — | — | — |
| | *Requester cache final status* | W[2] | R | R | I | R | W | L |
| **Write / Write-without-Fetch[3]** | *Requester cache to bus* | W, F | | | | W | — | — |
| | *Other cache initial status* | I | R | W | L[4] | R | — | — |
| | *Other cache final status* | I | I | I | L | I | — | — |
| | *Requester cache final status* | W | W | W | I | W | W | L |
| **Lock** | *Requester cache to bus* | W, F | | | | W | — | — |
| | *Other cache initial status* | I | R | W | L | R | — | — |
| | *Other cache final status* | I | I | I | L | I | — | — |
| | *Requester cache final status* | L | L | L | I | L | L | L[5] |
| **Unlock** | *Requester cache to bus* | U, S[6] | | | | — | — | U, S[6] |
| | *Requester cache final status* | A or I[7] | | | | R[7] | W[7] | W |
| **I/O Read/Write[8]** | *Request to bus* | W, F/S | | | | | | |
| | *Any cache initial status* | I | R | W | L[9] | | | |
| | *Any cache final status* | I | I | I | L | | | |
| **Memory Mode[8] Read/Write** | *Request to bus* | MM, F/S | | | | | | |
| | *Any cache initial status* | I | R[10] | W[10] | L[10] | | | |
| | *Any cache final status* | I | R | W | L | | | |
| *Source purges dirty block* | | | | | | | | |
| | *Requester cache to bus* | — | | | | Purge, S | Purge, S | — |
| | *Other cache initial status* | — | | | | R | — | — |
| | *Other cache final status* | — | | | | R,Src[11] | — | — |

## Table Notes

0. Abbreviations

| *Abbreviation* | *Meaning* |
|---|---|
| — | Not applicable |
| A | Absent |
| F | Fetch the block |
| I | Invalid (or absent) |
| L | Lock |
| MM | Memory mode read or write |
| R | Read privilege |
| S | Store the block (in memory) |
| Src | Source of latest version of block<br>• Location of clean/dirty status for the block<br>• When the block is requested by another cache, the source provides it and its current clean/dirty/lock status<br>• When purging the block, the source flushes it if dirty |
| U | Unlock |
| W | Write privilege |

1. It is a bug to attempt to read a block that is locked by another cache, unless hint read is implemented.
2. Write privilege is assumed if the block is invalid (or absent) in *all* other caches; otherwise read privilege is assumed.
3. On a hit (Read/Write/Lock status), write-without-fetch is handled by the cache like a normal write request. While on a miss (Invalid/Absent), the block is not fetched, hence the bus command is W, rather than W,F.
4. It is a bug to attempt to write a block that is locked by another cache.
5. It is a bug to attempt to lock a block that is already locked in your cache. This is because process-switching interrupts should be disabled when a busy-wait lock is locked, as explained in Section B.2.2, and a process should not lock a block again before unlocking it.
6. A bus access is made only if the lock is absent (had been purged) or if the state is lock-waiter. An operand may accompany (requiring the store), depending on the design.
7. It is a bug to attempt to unlock a block that is not locked in your cache.
8. The requester cache is equivalent to the other caches for I/O and memory mode commands.
9. It is a bug for I/O to be requested on a locked block.
10. If the object is non-cachable, such as an I/O port, it is a bug for the address to be valid in the cache.
11. The other cache takes source responsibility and clean/dirty the status for the block.

## Appendix 3

## Block-Invalidation Bus Traffic



### 1. General Concepts

After enumerating the components of bus traffic, along with several models, we will derive large upper bounds, and then smaller upper bounds, on the bus traffic of Goodman's (1983) invalidation write-through (Section B.2.3, Feature 4). Then we will briefly consider the bus traffic due to invalidating unshared data (Section B.2.3, Feature 5). Keep in mind that the purpose of this section is not to make highly accurate estimates, but simply to derive reasonable upper bounds in order to get a rough feel for the fraction of bus traffic generated by invalidation writes.

**Bus Traffic Components.** Under an update-by-block (write-back) cache policy, the bus traffic consists of the following components, some of which may not be present in particular systems. All except for the I/O transfers are generated by a processor's cache. The occasions for each are also indicated.

- *Block fetch from memory.* Occasions:
  - Read miss (unshared *vs.* shared data)
  - Write miss
  - Prefetch
- *Block fetch from another cache* (faster than fetch from memory). Occasions:
  - Read miss (unshared *vs.* shared data)
  - Write miss
  - Prefetch
- *Block flush to memory.* Occasions:
  - Miss
  - Prefetch
  - Process switch or termination
- *Block invalidation in other caches.* Occasions:
  - Write hit — if cache does not have write privilege
  - Write miss — in addition to block fetch, Goodman requires an invalidation write-through to memory

- *I/O transfer.* Occasions:
  - Input
  - Output

We will limit our attention to the following components, for simplicity, in deriving upper bounds:

- Block fetch from memory
  - Miss
- Block flush
  - Miss
- Block invalidation in other caches
  - Write hit
  - Write miss

Since we are restricting attention to only part of the total non-invalidation traffic (a denominator), the resulting values regarding cost will be upper bounds.

**Models.** We need to make basic assumptions about bus arbitration and memory structure. It seems to me that there are two reasonable models for bus arbitration and four for memory structure.

- Bus arbitration
  - *Word transfer:* Every word transfer requires arbitration.
  - *Multiword transfer:* The master can hold the bus through an arbitrarily long multiword transfer.
- Memory structure
  - 1. *Non-interleave; no address/data registers:* Memory has no address or data registers for servicing a read or write, so the master must hold and drive the address/data bus throughout the read or write for the full memory latency.
  - 2. (*Non-interleave; address/data registers:* Memory has address/data registers, so a *write* costs only *one* bus cycle. If, in addition, the bus allows split address/data phases for a read, a *read* costs only *two* bus cycles — one for the address and one for the data.)
  - 3. *External interleave; (address/data registers):* Memory is externally interleaved (each module is individually accessible to the processor), so it has address/data registers. A *write* costs *one* bus cycle, while a *read* costs *two*.
  - 4. *Internal interleave; (address/data registers):* Memory is internally interleaved (a module is not individually accessible to the processor), so on a block read, the address is distributed internally to all modules in parallel, taking less than one bus cycle per module.

Under the second memory model, if the bus does not allow split address/data phases for a read, this model simply reduces the cost of the invalidation write, as compared to the

first, so the first gives us an upper bound for the two models. If, on the other hand, the bus does allow split address/data phases, the second model becomes equivalent to the external interleave model in terms of bus traffic. For simplicity, then, let us ignore the second model.

Combining the two bus-arbitration models and the three resulting memory-structure models gives us three models of interest.

- *Low performance:* arb. every word, non-interleave, no adr/data registers
- *Medium performance:* arb. multiword, non-interleave, no adr/data registers
- *High performance:* arb. multiword, external interleave, (adr/data registers)
- *Very high performance:* arb. multiword, internal interleave, (adr/data registers)

## 2. Invalidation Write-Through: Large Upper Bounds

This topic emerges from Section B.2.3, Feature 4. Let use first consider just the *block fetch* and the *invalidation write on a write miss*, and in doing so we will derive the large upper bound $1/n_{words}$ on $Cost_{inval.wr}$, where $n_{word}$ is the number of bus words in a memory block. The invalidation on a write hit can be ignored in deriving an upper bound, since any scheme requires a bus access in this case, whereas only Goodman's scheme requires an extra access to invalidate on a write miss.

**Low/Med/High Performance Models.** In this context, let us define the following variables.

| *Variable* | *Number of ...* |
|---|---|
| $n_{words}$ | Bus data-words in a block |
| *Variable* | *Number of bus cycles required for ...* |
| $n_{arb}$ | Bus arbitration ($n_{arb} = 0$ if arb. is overlapped with prior transfer) (Only the successful arbitration is counted. An unsuccessful arbitration is counted in the cost of the processor that won it.) |
| $n_{lat}$ | Memory read latency for internal interleave model |
| $n_{read}$ | Memory read ($n_{read} = 2$ under external interleave model) |
| $n_{write}$ | Memory write ($n_{write} \leq n_{read}$; $n_{write} = 1$ under external interleave model) |
| $n_{fetch}$ | Fetching a block |

The fractional increase in bus traffic due to the invalidation write is the ratio of the total traffic in a system with that write to the total traffic in a system without that write, minus one. Since we are currently limiting our attention to the block fetch and the

invalidation write on a write miss, this increase, $Cost_{inval.wr}$, is as follows.

General:

$$Ratio_{traffic} = \frac{n_{fetch} + n_{inval.wr}}{n_{fetch}} > 1 \quad (1)$$

$$Cost_{inval.wr} = Ratio_{traffic} - 1 = \frac{n_{inval.wr}}{n_{fetch}} \quad (2)$$

We see that the cost fraction is just the ratio of the invalidation write traffic, to the traffic in a system without those writes.

The subsequent equations indicate the value of $Cost_{inval.wr}$ under each model. The cost under the high performance (external interleave) model is just the cost under the medium performance model with $n_{read} = 2$, $n_{write} = 1$; while the internal interleave model reduces to the external interleave model if $n_{lat} = n_{words}$.

Low perf. ($n_{write} \leq n_{read}$, $n_{write} \approx n_{read}$):

$$Cost_{lo.perf} = \frac{(n_{arb} + n_{read})\, n_{words} + (n_{arb} + n_{write})}{(n_{arb} + n_{read})\, n_{words}} - 1$$

$$= \frac{n_{arb} + n_{write}}{(n_{arb} + n_{read})\, n_{words}} \quad (3)$$

Medium perf. ($n_{write} \leq n_{read}$, $n_{write} \approx n_{read}$):

$$Cost_{med.perf} = \frac{(n_{arb} + n_{read} n_{words}) + n_{write}}{n_{arb} + n_{read} n_{words}} - 1 = \frac{n_{write}}{n_{arb} + n_{read} n_{words}} \quad (4)$$

High perf. — external interleave ($n_{write} = 1$, $n_{read} = 2$):

$$Cost_{hi.perf} = Cost_{med.perf} \mid (n_{read} = 2, n_{write} = 1) = \frac{1}{n_{arb} + 2n_{words}} \quad (5)$$

Very high perf. — internal interleave ($n_{write} = n_{read} = 1$, $n_{lat} \leq n_{words}$):

$$Cost_{v.hi.perf} = \frac{n_{write}}{n_{arb} + n_{lat} + n_{read} n_{words}} = \frac{1}{n_{arb} + n_{lat} + n_{words}} \quad (6)$$

Now consider two observations, assuming $n_{write} \approx n_{read} \geq 2$, $n_{lat} \leq n_{words}$, where relevant.

Order:

$$Cost_{hi.perf} \leq Cost_{v.hi.perf} \leq Cost_{med.perf} \leq Cost_{lo.perf} \leq 1/n_{words}$$

Overlapped arb. ($n_{arb} = 0$):

$$2Cost_{hi.perf} \approx Cost_{med.perf} \approx Cost_{lo.perf} \approx 1/n_{words}$$

Regarding order, $Cost_{hi.perf} < Cost_{v.hi.perf}$ to the extent that $n_{words} > n_{lat}$. $Cost_{hi.perf} < Cost_{med.perf}$ to the extent that $n_{write} \approx n_{read} > 2$. $Cost_{med.perf} < Cost_{lo.perf}$ to the extent that $n_{arb} > 0$, assuming $n_{read} \approx n_{write}$. And $Cost_{lo.perf} < 1/n_{words}$ to the extent that $n_{write} < n_{read}$.

We see that the ordering implies the large upper bound mentioned at the outset:

Large upper bound:

$$Cost_{inval.wr} \leq 1/n_{words}$$

In addition, under overlapped arbitration, the costs of the low and medium performance models are about the same, namely $1/n_{words}$, since $n_{write} \approx n_{read}$. Their cost is also twice that of the high performance model.

## 3. Invalidation Write-Through: Small Upper Bounds

Having examined large upper bounds for the cost, let us turn our attention to estimating the cost more closely. Specifically, let us look at the invalidation write in the context, not only of block fetches and invalidations, but also in the context of flushes.

**Variables.** With this in mind, let us define the following variables, some of which we have already seen. It is assumed that only data is written — the code does not modify itself.

| *Variable* | *Number of ...* |
|---|---|
| $n_{words}$ | Bus data-words in a block |
| *Variable* | *Number of bus cycles required for ...* |
| $n_{arb}$ | Bus arbitration ($n_{arb} = 0$ if arb. is overlapped with prior transfer) (Only the successful arbitration is counted. An unsuccessful arbitration is counted in the cost of the processor that won it.) |
| $n_{read}$ | Memory read ($n_{read} = 2$ under external interleave model) |
| $n_{write}$ | Memory write ($n_{write} \leq n_{read}$; $n_{write} = 1$ under external interleave model) |
| $n_{fetch}$ | Fetching a block |
| $n_{flush}$ | Flushing a block |
| $n_{inval.wr}$ | An invalidation write ($n_{inval.wr} = 0$ in a scheme without the write) |
| *Variable* | *The probability that a processor memory reference causes ...* |
| *(Note: '&' represents logical conjunction, or equivalently, set intersection.)* | |
| $p_{miss}$ | A miss |
| $p_{read\&miss}$ | A read miss |
| $p_{write\&miss}$ | A write miss ($p_{read\&miss} + p_{write\&miss} = p_{miss}$) |
| $p_{data\&miss}$ | A data read or write miss |
| $p_{data\&read\&miss}$ | A data-read miss ($p_{data\&read\&miss} + p_{write\&miss} = p_{data\&miss}$) |
| $p_{ins\&miss}$ | An instruction miss ($p_{ins\&miss} + p_{data\&read\&miss} = p_{read\&miss}$) |
| $p_{dirty\&miss}$ | A dirty miss, that is, a dirty block must be replaced on the miss |

**Equations.** The ratio of mean miss traffic in a system with the invalidation write, to mean miss traffic in a system without the write, is the basis from which $Cost_{inval.wr}$, or the fractional increase in traffic due to the invalidation write, is calculated.

$$Ratio_{traffic\&miss} = \mu_{traffic(inval.wr)\&miss} \; / \; \mu_{traffic(no.inval.wr)\&miss} \tag{7}$$

$$Cost_{inval.wr} = Ratio_{traffic\&miss} - 1 \tag{8}$$

The ratio may be less than one, it turns out. This is because the invalidation-write scheme can produce less traffic than the other by reducing the number of dirty blocks, since the invalidation write makes the block clean, though written once. So if the block is purged before being written again, it will not need to be flushed. The negativity of the cost, thus, shows the fractional *decrease* in traffic due to the invalidation write.

The mean miss traffic generated by the processor on a miss consists of the following:

$$\mu_{traffic(X)\&miss} = p_{read\&miss} n_{fetch} + p_{write\&miss}( n_{fetch} + n_{inval.wr} ) + p_{dirty(X)\&miss} n_{flush}$$

$$= p_{miss} n_{fetch} + p_{write\&miss} n_{inval.wr} + p_{dirty(X)\&miss} n_{flush} \tag{9}$$

where $X = inval.wr$ (invalidation-write scheme) or $no.inval.wr$. In the former case, $n_{inval.wr} > 0$, while in the latter $n_{inval.wr} = 0$.

The ratio and cost expanded are

$$Ratio_{traffic\&miss} = \frac{p_{miss} n_{fetch} + p_{write\&miss} n_{inval.wr} + p_{dirty(inval.wr)\&miss} n_{flush}}{p_{miss} n_{fetch} + p_{dirty(no.inval.wr)\&miss} n_{flush}} \tag{10}$$

$$Cost_{inval.wr} = \tag{11}$$

$$\frac{p_{write\&miss} n_{inval.wr} + ( p_{dirty(inval.wr)\&miss} - p_{dirty(no.inval.wr)\&miss} ) n_{flush}}{p_{miss} n_{fetch} + p_{dirty(no.inval.wr)\&miss} n_{flush}}$$

In general, the number of dirty blocks under an invalidation-write system may be less than the number of dirty blocks under an alternative system, making $p_{dirty(inval.wr)\&miss} \leq p_{dirty(no.inval.wr)\&miss}$. We are interested in an upper bound on the cost when the traffic ratio is greater than one (the invalidation-write system is worse), so we can simplify by assuming the two are equal. Then the traffic ratio will be greater than one, and the fractional increase in traffic due to the invalidation write will be positive.

For the sake of simplicity, now, let us consider just two extreme models: the low performance model and the high performance model, for these yielded the largest and smallest upper bounds on $Cost_{inval.wr}$, respectively.

- *Low performance:* arbitrate every word, non-interleave, no address/data registers
- *High performance:* arbitrate multiword, external interleave, (address/data registers)
  - Assume just one arbitration for a flush, fetch, and invalidation write

In the high performance model, we will assume that if a flush or invalidation write is necessary, neither requires another bus arbitration. Instead, the flush, fetch, and invalidation write will run back-to-back, requiring just one arbitration for all.

The following equations, then, emerge. Notice that the bus traffic at a fetch and an invalidation write are as before (Equations 3,5).

Low perf:

$$n_{fetch(lo.perf)} = ( n_{arb} + n_{read} ) n_{words} \tag{12}$$

High perf:

$$n_{fetch(hi.perf)} = n_{arb} + 2n_{words} \tag{13}$$

Low perf:

$$n_{inval.wr(lo.perf)} = n_{arb} + n_{write} \tag{14}$$

High perf:

$$n_{inval.wr(hi.perf)} = 1 \tag{15}$$

Regarding the flush traffic, in the low performance system, the flush traffic is the same as the fetch traffic, while in the high performance system, a flush requires just one bus cycle for every word, due to the interleaving.

Low perf:

$$n_{flush(lo.perf)} = n_{fetch(lo.perf)} \tag{16}$$

High perf:

$$n_{flush(hi.perf)} = n_{words} \tag{17}$$

Since we are interested in an upper bound on $Cost_{inval.wr}$, we can simplify, as mentioned, by eliminating the potential advantage of the invalidation write, and let $p_{dirty(inval.wr)\&miss} = p_{dirty(no.inval.wr)\&miss}$, giving us

$$Cost_{inval.wr} = \frac{p_{write\&miss} n_{inval.wr}}{p_{miss} n_{fetch} + p_{dirty\&miss} n_{flush}} \quad (18)$$

For the low performance model, $n_{fetch} = n_{flush}$, $n_{inval.wr}/n_{fetch} = 1/n_{words}$, giving us

$$Cost_{lo.perf} = \frac{p_{write\&miss}}{(p_{miss} + p_{dirty\&miss}) n_{words}} \quad (19)$$

For the high performance model, $n_{inval.wr} = 1$, $n_{flush} = n_{words}$, and $n_{fetch} = n_{arb} + 2n_{words}$, yielding,

$$Cost_{hi.perf} = \frac{p_{write\&miss}}{p_{miss} (n_{arb} + 2n_{words}) + p_{dirty\&miss} n_{words}} \quad (20)$$

**Numerical Calculations.** In order to calculate this value we would like $p_{dirty\&miss}$, $p_{write\&miss}$, and $p_{miss}$ from the same cache and set of traces, with cache size varied. My source for this kind of information is the work of Professor Alan Smith, but the desired data is not available in the papers that I have. (See the references.) However, my persistent efforts were not entirely unsuccessful, for I was able to locate three sources from which I could derive the desired values. The sources do not represent the same cache and set of traces. Yet the values range widely from trace to trace on the same cache, anyway, so even if they had the same source, we would still be looking at a *wide range* of values. And that range would probably have a large intersection with the range that I do have.

In Smith (1979, p. 24) I discovered $p_{dirty \mid miss} = .18$ to .56, in an 8K-byte, 4-way associative, instruction-and-data cache having 32-byte blocks. This is the fraction of replaced blocks that were dirty. (Smith, 1985, shows the fraction of dirty blocks $p_{dirty}$ in a 16K-byte, fully-associative data cache to range from .20 to .80.)

From data in Smith (1984, p. 17.1 - 17.2), I derived $p_{write \mid (data\&miss)} = .3$ to .5 for a 4K-byte, 2-way associative, data cache having 16-byte blocks. This is the probability that a data miss is a write miss, and is derived from $<p_{write\&miss \mid data}, p_{miss \mid data}>$ pairs by dividing the first by the second. I conditionalized on $p_{data\&miss}$ in order to use <instruction, data> miss-rate pairs, described next.

In Smith (1982, fig. 25-28) I discovered $p_{miss \mid ins}$ and $p_{miss \mid data}$ pairs for a range of instruction and data cache sizes starting with a 2K-byte, direct-mapped cache and

continuing beyond a 64K-byte, 32-way associative cache. (The number of sets was held constant at 64, and block size was maintained at 32 bytes. Keep in mind that after set size reaches eight, it has little further effect on decreasing the miss rate. This was shown earlier in Smith's paper. Also, for each size value, the instruction and data caches were the same size.) I took my values from Figure 25: $p_{miss\,|\,ins} = .004$ to .05, $p_{miss\,|\,data} =$ .02 to .05. The two variables are not independent, but are paired along the two ranges shown.

Finally, Smith (1985) shows that data references comprise about .25 to .50 of the memory references for microprocessor and mini/mainframe computers, respectively; that is, $p_{data} = .25$ to .50, $p_{ins} = .75$ to .50.

Assuming that it is meaningful to relate the values from the different situations, since their ranges are so large, we can derive the values we need.

$$\begin{aligned} p_{miss} &= p_{ins\&miss} + p_{data\&miss} \\ &= p_{miss\,|\,ins}\ p_{ins} + p_{miss\,|\,data}\ p_{data} \end{aligned} \tag{21}$$

$$p_{dirty\&miss} = p_{dirty\,|\,miss}\ p_{miss}$$

$$p_{write\&miss} = p_{write\&data\&miss} \quad \text{(assuming only data blocks are written)}$$

$$= p_{write\,|\,(data\&miss)}\ p_{data\&miss}$$

With these values in hand, I selected a range of the variables of interest, and calculated the costs for the low performance and high performance models. Since *Cost* increases in $p_{data}$ (the numerator increases faster than the denominator), and since we are interested in upper bounds, for simplicity I let $p_{data} = .50$. The results are shown in Table 1. We can see that all cost values are well below the upper bound of $1/n_{words}$.

**Table 1. Approximate Upper Bounds on the Bus Traffic due to Invalidation Write-Through**

*(Based on a range of values derived from data in Smith 1978, 1982, 1984, 1985. $p_{data} = .50$. '•' means "same as above.")*

| $n_{words}$ | $n_{wt}$ | $p_{dirty \mid miss}$ | $p_{write \mid (data \& miss)}$ | $p_{miss \mid ins}$ | $p_{miss \mid data}$ | $p_{data \& miss}$ | $p_{miss}$ | $p_{dirty \& miss}$ | $p_{write \& miss}$ | Cost (%) lo.perf | Cost (%) hi.perf | $\frac{1}{n_{words}}$ (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | .2 | .3 | .0045 | .02 | .01 | .0123 | .0001 | .003 | 24 | 12 | |
| 1 | 0 | .2 | .5 | .0045 | .02 | .01 | .0123 | .0001 | .005 | 40 | 20 | |
| 1 | 0 | .2 | .3 | .05 | .05 | .025 | .05 | .01 | .0075 | 13 | 6.8 | |
| 1 | 0 | .2 | .5 | .05 | .05 | .025 | .05 | .01 | .0125 | 21 | 11 | |
| 1 | 0 | .5 | .3 | .05 | .05 | .025 | .05 | .025 | .0075 | 10 | 6.0 | |
| 1 | 0 | .5 | .5 | .05 | .05 | .025 | .05 | .025 | .0125 | 17 | 10 | |
| | | | | | | | | | | | | 100 |
| | 3 | | | | | | | | | | 4.8 | |
| | 3 | | | | | | | | | | 8.1 | |
| • | 3 | • | • | • | • | • | • | • | • | • | 2.9 | |
| | 3 | | | | | | | | | | 4.8 | |
| | 3 | | | | | | | | | | 2.7 | |
| | 3 | | | | | | | | | | 4.6 | |
| 2 | 0 | | | | | | | | | 12 | 6.1 | |
| 2 | 0 | | | | | | | | | 20 | 10 | |
| 2 | 0 | • | • | • | • | • | • | • | • | 6.3 | 3.4 | |
| 2 | 0 | | | | | | | | | 10 | 5.7 | |
| 2 | 0 | | | | | | | | | 5.0 | 3.0 | |
| 2 | 0 | | | | | | | | | 8.3 | 5.0 | |
| | | | | | | | | | | | | 50 |
| | 3 | | | | | | | | | | 3.5 | |
| | 3 | | | | | | | | | | 5.8 | |
| • | 3 | • | • | • | • | • | • | • | • | • | 2.0 | |
| | 3 | | | | | | | | | | 3.4 | |
| | 3 | | | | | | | | | | 1.9 | |
| | 3 | | | | | | | | | | 3.2 | |
| 4 | 0 | | | | | | | | | 6.0 | 3.0 | |
| 4 | 0 | | | | | | | | | 10 | 5.1 | |
| 4 | 0 | • | • | • | • | • | • | • | • | 3.1 | 1.7 | |
| 4 | 0 | | | | | | | | | 5.2 | 2.8 | |
| 4 | 0 | | | | | | | | | 2.5 | 1.5 | |
| 4 | 0 | | | | | | | | | 4.2 | 2.5 | |
| | | | | | | | | | | | | 25 |
| | 3 | | | | | | | | | | 2.2 | |
| | 3 | | | | | | | | | | 3.7 | |
| • | 3 | • | • | • | • | • | • | • | • | • | 1.3 | |
| | 3 | | | | | | | | | | 2.1 | |
| | 3 | | | | | | | | | | 1.2 | |
| | 3 | | | | | | | | | | 1.9 | |
| 8 | 0 | | | | | | | | | 3.0 | 1.5 | |
| 8 | 0 | | | | | | | | | 5.0 | 2.5 | |
| 8 | 0 | • | • | • | • | • | • | • | • | 1.6 | .85 | |
| 8 | 0 | | | | | | | | | 2.8 | 1.4 | |
| 8 | 0 | | | | | | | | | 1.3 | .75 | |
| 8 | 0 | | | | | | | | | 2.1 | 1.3 | |
| | | | | | | | | | | | | 12.5 |
| | 3 | | | | | | | | | | 1.3 | |
| | 3 | | | | | | | | | | 2.1 | |
| • | 3 | • | • | • | • | • | • | • | • | • | .73 | |
| | 3 | | | | | | | | | | 1.2 | |
| | 3 | | | | | | | | | | .85 | |
| | 3 | | | | | | | | | | 1.1 | |

## 4. Invalidation of Unshared Data

This issue emerges from Section B.2.3, Feature 5. As noted there, upper bounds can be obtained by assuming that all data is unshared, all misses are read misses, and all data blocks are written. In this case, a scheme that does not fetch unshared data for write privilege on a read miss will require a later bus access for every block that is fetched in order to invalidate the block in other caches. Consequently, an upper bound on the frequency of these invalidations is the data miss-rate $p_{data\&miss}$.

A smaller upper bound on the frequency of invalidating unshared data is obtained by dropping the last two assumptions, retaining just the first — all data is unshared. Suppose, in addition, this data always arrives clean, as it does if fetched from memory, as in a uniprocessor system. Then an upper bound on the frequency of invalidations in a scheme that does not fetch unshared data for write-privilege on read misses is the frequency of a write hit to a clean block $p_{write\&hit\&clean}$. I derive this value as follows, since I am not aware of any direct measurement of it.

$$p_{write\&hit\&clean} = p_{write\&clean} - p_{write\&miss\&clean} \tag{22}$$

$$= p_{miss}\, p_{dirty} - p_{write\&miss} \tag{23}$$

$$= p_{data\&miss}\, p_{dirty\,|\,data} - p_{write\&miss} \tag{24}$$

(Equation 23 or 24 is used to calculate $p_{write\&hit\&clean}$, according to the data that is available.) The equality of the first terms of the right-hand sides becomes evident from the following identities.

$$p_{miss}\, p_{dirty} = \frac{\#\text{ block fetches}}{\#\text{ mem. ref.s}} \cdot \frac{\#\text{ blocks written}}{\#\text{ block fetches}} \tag{25}$$

$$= \frac{\#\text{ blocks written}}{\#\text{ mem. ref.s}} = p_{write\&clean}$$

The number of blocks written, here, really means the number of block fetches in which the block is subsequently written before being purged, while $p_{dirty}$ is the fraction of dirty blocks. Equation 25, then, accounts for Equation 23, while Equation 24 is verified in an equivalent way by restricting attention to data blocks, and by assuming that only data blocks are written (the code does not modify itself). The variable $p_{dirty\,|\,data}$ is the fraction of data blocks that are dirty.

Having derived the frequency of invalidations, note that the invalidation itself will cost $n_{arb} + 1$ bus cycles.

Upper bounds on $Cost_{inval}$ follow by replacing the numerator in the earlier expressions of $Cost_{inval.wr}$ by $p_{write\&hit\&clean}\,(n_{arb} + 1)$, (Equations 18,19,20). I use Equation 24 to calculate $p_{write\&hit\&clean}$ since Smith (1985) provides values of $p_{dirty\,|\,data}$ (for unshared data) by purging the entire data cache periodically. These values range from .2 to .8, but tend toward .5.

Rather than consider a whole new table comparable to Table 1, for simplicity let us take the ratio of the numerators, which are independent of $n_{words}$, and this will give us the ratio of the costs. Table 2, then, shows resulting values, along with the quotient $Cost_{inval}$ / $Cost_{inval.wr}$, which is obtained from the ratio of the two numerators, and allows $Cost_{inval}$ to be conveniently derived from $Cost_{inval.wr}$ in Table 1. Notice that $p_{dirty \mid data}$ is shown for both the central value .5 and the largest value .8. The latter is included since we are interested in upper bounds on the cost.

**Table 2. Values Relevant to Determining Approximate Upper Bounds on Bus Traffic for Invalidating Unshared Data**

*(Based on values from Table 1. '*' means "same as above.")*

| $p_{dirty \mid data}$ | $p_{data\&miss}$ | $p_{write\&miss}$ | $p_{write\&hit\&clean}$ | $\frac{Cost_{inval}}{Cost_{inval.wr}}$ |
|---|---|---|---|---|
| .5 | .01 | .003 | .0020 | .67 ($n_{arb}$+1) |
| .5 | .01 | .005 | 0 | 0 |
| .5 | .025 | .0075 | .0050 | .67 ($n_{arb}$+1) |
| .5 | .025 | .0125 | 0 | 0 |
| .8 | | | .0050 | 1.67 ($n_{arb}$+1) |
| .8 | * | * | .0030 | .60 ($n_{arb}$+1) |
| .8 | | | .0125 | 1.67 ($n_{arb}$+1) |
| .8 | | | .0075 | .60 ($n_{arb}$+1) |

According to Table 2, for *overlapped arbitration* ($n_{arb}$=0), approximate upper bounds on $Cost_{inval}$ can usually be derived by multiplying the values $Cost_{inval.wr}$ in Table 1 by 2/3. So the fraction of bus traffic for invalidating unshared data tends to be well under $1/n_{words}$ in this case. When multiplying by 1.67, the largest ratio of Table 2, the resulting values are still well under $1/n_{words}$.

The small amount of bus traffic required for invalidating unshared data in this case is also evident in the simulations of Archibald and Baer (1985). In handling unshared four-word blocks, the Papamarcos and Patel (1984) scheme, which fetches unshared data for write privilege, has only a very small advantage over the naive protocol of Katz et al. (1985), which does not fetch unshared data for write privilege.

Finally, we see, the larger $n_{arb}$, the greater the cost of invalidating unshared data.

## Appendix 4

## Non-Broadcast Cache Systems



### 1. Cache Synchronization

The split-level Aquarius architecture has a broadcast-cache system upstairs and a non-broadcast system downstairs, so we are concerned with efficient cache synchronization in both kinds of system (Section D.2). Although the non-broadcast system will hold no hard atoms, for completeness let us examine the issues of a general non-broadcast cache system in which hard atoms can reside.

From Section B.1.3, we recall the two hardware tasks in cache synchronization.

- *Serialize conflicting access requests:* hard atoms only
- *Provide the latest version of requested block:* all objects

We will now look at hard atoms and all other writable objects, respectively.

#### *Hard Atoms*

Let us make several reasonable simplifying assumptions.

- *Fast read-modify-write:* A hard atom is used for fast atomic read-modify-write operations.
- *Single block:* A hard atom is contained entirely on a single memory block.
- *Busy wait:* Busy wait is less costly than sleep wait for this case, so is preferable.
- *Contention:* Contention is great enough that the same processor rarely accesses the same hard atom twice in a row before another processor accesses the atom.

It follows that hard atoms can be locked simply by using a lock bit on the memory block of the atom. This is similar to the memory-bit backup for the cache lock-state in a broadcast protocol (Section B.2.2), except that the lock bits here are not cached, and a hard atom is *purged*, as well as unlocked, at the last write to it. The memory unit, cache, or processor will need to execute an atomic read-and-set on the lock bit as the data is fetched. It would be fastest to have memory execute the read-and-set, and the requisite circuit for this is very simple. Barring this slight complication to memory, though, the cache should execute the read-and-set (for the sake of speed), again using a very simple, fast circuit just for that operation.

This simple scheme, then, fulfills both synchronization requirements for hard atoms — serialization of requests, and provision of the latest version of the data.

*Other Writable Objects*

Soft atoms and unshared writable data present us with two cases for providing the latest version of writable data, as cited in Section B.1.2.

- *Two different processes* on two different processors access the same writable, shared data (soft atom)
- *One process* on two different processors accesses the same writable (shared or unshared) data

Let us consider each in turn.

**Two Different Processes.** Providing the latest version of a soft atom, as for a hard atom, can be insured by purging all blocks of the atom from the cache after each use of it, flushing the dirty blocks to memory. The next access to any block of the atom by that CPU (or, through generalization, by any other CPU) will then generate a cache miss — due to the *purge* — causing the block to be fetched from memory, where its latest version will reside — due to the *flush*.

To be more specific, recall that conflicting access requests for a soft atom are serialized by the software using a synchronization descriptor. So let us assume that a soft atom occupies a range of sequential addresses, and that this range is recorded in the atom's synchronization descriptor. When a process is about to release access (read or write privilege) to the soft atom, it runs through the block addresses in the address range, informing the cache to purge each block. At any such hit, the cache purges the block, which implies flushing the block to memory if dirty (Figure 29). Each such block is devoted to the atom so that this flushing will not overwrite another atom on the same block (Figure 30).

**One Process.** In addition, when a process goes to sleep, it must be awakened on the same CPU by being put in a ready queue for that CPU. Or else, following the above idea, when the process goes to sleep, it must purge *all* writable data, not only shared data that is currently being accessed, but unshared data as well. There are two ways to implement this: distinguish writable and read-only data and keep a record in the cache directory, or else purge the entire cache when going to sleep.

**Version numbers.** An alternative to purging soft atoms in the above two situations would be to maintain, in the synchronization descriptor of a soft atom, the number of times the atom has been written, incrementing the number each time write privilege is released. This number, the *version number,* would be recorded in the cache directory for a block of the atom when the block is fetched into the cache. On an address hit, the descriptor version number would be compared with the version number recorded for the block in the cache, and if the two are equal, the block need not be fetched from memory. In addition, when the process is about to release write privilege to the soft atom, it increments the version number in the descriptor. It then runs through the block addresses in the atom address range, and at any such hit, instead of telling the cache to purge the block, as in the previous scheme, tells the cache to record the new version number in the

directory entry for the block; and if the block is dirty, the cache will flush it. The cost of this method, however, may well outweigh the performance gain of fewer fetches. Both the cost and the performance gain must be evaluated in greater detail for a specific system in order to determine which outweighs the other.

**Presence List.** A presence list (Section B.1.3) could be used for writable data. But if there is no broadcast capability, when a block must be purged, the caches in which it resides must be contacted *sequentially*. On the surface, this appears too slow for data that may reside in more than one cache.

Archibald and Baer (1984), nevertheless, do not believe that sequential notification is too slow, since they feel that the occasions for using it will be sufficiently infrequent. In fact, surprisingly they eliminate the presence list, maintaining a simple four-state indicator in memory: (1) absent from caches, (2) may have read privilege in some cache, (3) has read privilege in exactly one cache, (4) has write privilege and dirty status in exactly one cache. Consequently, when sequential notification is required, *all* caches must be notified since there is no presence list. (Note that although Archibald and Baer use the term 'broadcast' in their paper, they actually mean that the recipients of the message are notified sequentially, not concurrently.)

## 2. Efficient Busy Wait

Although this topic is not relevant to the Aquarius architecture, since efficient busy wait will be implemented in a broadcast system (the synchronization system) I would, nevertheless, like to explore it here, for completeness. Three methods of busy wait for non-broadcast systems will be presented.

- Immediate retry
- Fixed-delay retry
- FIFO retry

The two simpler methods do less to reduce memory accesses, so are sufficient for lower contention locks, while the most structured method may be useful for high contention locks. If the hardware does not implement sleep/priority queues, the high contention locks will undoubtedly be the locks on the sleep-wait queues and service-request queues. How much contention must a lock receive in order to warrant the highly-structured wait scheme? This can only be determined by detailed performance analysis, which is not provided here.

In a non-broadcast system, the hard atom containing the status information on a lock must be *polled* in shared-memory using the switch. It is desired, then, to minimize the number of unsuccessful polls or retries. If, in addition, the processor is idling while waiting, it is desired to minimize the idle time, by polling as soon after the lock is released as possible. However, the time of release can only be estimated, so to poll as soon after release as possible it would generally be necessary to poll several times unsuccessfully. *A*

*tradeoff*, consequently, emerges:

- Minimize the *number of polls*

*vs.*

- Minimize the *time from release to poll.*

Let us, then, look at the three polling disciplines.

**Immediate Retry.** Under immediate retry, the process retries immediately after a failure — the process spins in a loop dedicated to testing the lock. This is the simplest but worst policy. For though it will catch the lock at the earliest possible moment, it generates the most unsuccessful polls, flooding the switch with useless accesses, and disallowing the CPU from doing any other work.

**Fixed-Delay Retry.** In fixed-delay retry, the process retries after a fixed delay, the same delay for all processes. The delay is usually half of the time that a process is expected to hold the lock (Denning, Dennis, Brumfield 1981). This is better, but does not take into account contention for the lock — the number of processes that are waiting on the lock at the time the lock is tested.

**FIFO Retry.** Under FIFO retry, at the time of a poll, the process schedules its subsequent poll based on the number of waiting processes ahead of it. This number is determined using a sequencer, and the appropriate delay is then determined using this number and a time value. This method of busy wait is quite complicated, and may not be feasible. Nevertheless, for the sake of exploration, let us plunge onward.

*FIFO Queuing.* A *sequencer* is data structure that contains two hard atoms — a *ticket* field and a *next* field — and is used to manage access by processes to a resource pool using busy wait. The waiting processes constitute a busy-wait queue, in contrast to the semaphore sleep-wait queue, because the processes are busy waiting rather than sleep waiting in the queue. If the maximum size of the pool is $n$, then the current size varies between 0 and $n$ as the items are allocated and deallocated. The ticket is initialized to zero, the next field is initialized to $n$, and each is operated on by increment. The resulting difference between the two values indicates the following, where *size* refers to current size of the pool or queue.

- Next−ticket $\geq 0$ indicates
  - *Pool size* — number of resource items available
  - *Queue empty* — no processes waiting
- Next−ticket $\leq 0$ indicates
  - *Pool empty* — no resource items available
  - *Queue size* — number of processes waiting

When a process first accesses the sequencer, ticket is atomically read and incremented (the process takes a ticket), and then next is read (the process finds out who is next). If

the two values are not covered by the same lock, next is read after taking a ticket, since next may be incremented in the meantime, and the process should get the latest value possible. The process then subtracts its ticket value from next, yielding the number of resource items available. If the number is greater than zero, the process has gained access to the pool; otherwise it must wait, and poll later. At the time of retry, the process reads just next, and, as before, subtracts its ticket value from next, yielding the number of resource items available. When a process releases access, next is incremented atomically.

The situation of interest here is to organize busy waiting on a lock which grants sole access to shared data, such as a sleep-wait queue. Consequently, $n$ is just one, and we will speak of the process having access as holding a lock.[19]

*Retry Delay.* Waiting processes schedule their retries to a lock as follows. The length of time that a process is expected to hold the lock is kept in an additional *time* field with the sequencer. Noting that the difference already calculated, next—ticket, indicates the number of processes waiting ahead of the current process, the time and difference (with the correct sign) are multiplied, adding on, say, half of the time, to cover for the current lock holder. The process will then retry at that time. If exact multiplication is too slow, then some reasonable approximation is made.

A stochastic model is needed to determine what the time value and delay function should be, due to the *variability* involved. The variability in the time that a process holds a lock is due to conditional branches, switch accesses, and traps. The variability in waiting for switch access can be reduced if a processor temporarily boosts its switch arbitration priority while holding a lock. The atomic operation could also be executed by the CPU holding onto the switch throughout the operation, but for a lengthy operation this would be wasteful of the switch. Process-switching interrupts are disabled when holding a lock (as discussed in Section B.2.2); while other traps may remain enabled, according to the judgment of the process. But variability also occurs in the time between the release of a lock and the moment that the next process locks the lock, as well as in the time between one poll of a waiting process and the subsequent poll of the same process. (Remember that a process must get access to the switch in order to poll.) Just the same, each poll does obtain the current value of next, so the new difference, next—ticket, can be calculated, and *the subsequent poll will not be thrown off by the variability up to this point.* This is a critical property of the sequencer.

*Disabling Process-Switching.* Under FIFO busy wait, *a process disables process-switching interrupts while waiting, even if it must idle.* The reasons for this are two.

[19] Reed and Kanodia (1979) discuss the use of sequencers to implement sleep wait rather than busy wait. However, using sequencers to implement sleep wait appears to me to add needless complexity as compared to using semaphores, whereas sequencers add valuable complexity for implementing busy wait. The discussion of Reed and Kanodia seems to me to be mainly of theoretical interest.

First, the variability in the wait time would otherwise be too great to make the delay calculation useful. Second, when a process's turn arrives, that process has, in effect, locked the data — no other process can access the data — so it should be ready to run at that time. Under busy wait, this implies that the process does not allow itself to be put on a ready queue while waiting. What, then, are the effects of disabling process-switching?

Suppose all processes busy-waiting on a particular lock must *idle* while waiting. Their disabling of process-switching interrupts forces their processors to idle during that time. Would it be better to forget the FIFO retry scheme and revert to fixed-delay, which does not require the disabling of process-switching interrupts? This depends on how much less idling (a positive effect) and how many more retries (a worse negative effect) are expected *per processor* under fixed delay. The overall effect on the system, in turn, depends on the *percentage of processors* that are expected to be busy-waiting on the lock at any one time. These effects can be determined using a stochastic model and simulation.

Further, suppose an *urgent interrupt* is ignored while busy-waiting. What are the effects of this? It turns out that this is only a problem for I/O processors which handle real-time interrupts, not CPUs. These considerations and others are discussed in Appendix 5.

Finally consider that if process-switching interrupts were, for some reason, enabled during FIFO busy wait, and if the process were indeed switched out, its priority should temporarily be raised. For, as in holding a lock, this is necessary not only for efficiency, but also in order to avoid deadlock; otherwise a lower priority process could be ahead of a higher priority process in the busy-wait queue and never again get processor time after it is put on a prioritized ready queue.

**Time Measure.** Polling requires some measure of time so that a process can estimate when to retry (except under immediate retry). Let us consider a few possible measures. One measure is the number of processor instructions executed during the time period of interest, but this measure is very crude, since different instructions take different amounts of time. The finest measure is the number of clock cycles occurring during the interval of time. While an intermediate measure is some fraction of the actual number of clock cycles. The advantage of using a fraction is that it will allow smaller time fields and hence less memory space and less comparison hardware. Finally, remember that no matter what time measure is used, a stochastic model for the various components of the wait interval is needed, in order to account for the variability that will occur.

## Appendix 5

## Interrupt Management

At various times throughout the discussion we have met occasions for generating and for disabling specific types of interrupts. Now let us gather together the issues surrounding interrupts, and look at them in the detailed context thus far developed. We are concerned, here, with interrupts, not exceptions, since *exceptions* are internal to a program, and thus the effects of disabling them are relatively clear to the program, and their disabling will simply affect the processing of the program that does the disabling. Whereas some *interrupts* are generated from outside the program, so the effects of disabling them reach into the rest of the system and are not so clear cut. Interrupts should be considered more closely, then, to elucidate the effects of their being disabled.

### *Interrupt Types*

A computer system may be thought of as executing two kinds of activity: computation and I/O. In an efficient multiprocessor system, special processors (I/O processors) are devoted to I/O so that they can meet the I/O demands without taking processor time from the computation. The rest of the processors (CPUs) are devoted to computation, and their processes can, conversely, disable process-switching interrupts without concern for ignoring real-time I/O signals. These CPU processes issue I/O requests for the I/O processors (IOPs) to service, receive the replies from the IOPs, and manage the buffers that are used by the IOPs in main memory; but the CPU processes remain detached from the real-time constraints of the I/O. This approach, in fact, generally holds for so-called uniprocessor (single CPU) systems, whose IOPs may range from powerful programmable processors to simple finite-state machines.

With this distinction in mind, let us consider the interrupts that a CPU and an IOP may encounter. The following two lists are the beginning of such an enumeration.

- CPU interrupts
  - *Priority preemption:* In a broadcast system, a process is moved from a wait queue (or a CPU) to a ready queue, preempting a lower priority process from its CPU.
  - *Quantum timer:* In a multiprogrammed system, a process has run for its entire time slice or quantum and must give up its CPU.
  - *Busy-wait termination:* In a broadcast system, a busy-waiting process can now take its turn and access the data.
  - *Routine operating system tasks:* Interrupts, such as time-of-day interrupts, will cause a temporary switch to the operating system, and then back again to the user process.
  - *Hardware errors:* Processor, switch, memory, power failure, soft reset.
- IOP interrupts
  - *Ready/Done:* An interrupt from an I/O device that has just become ready or has just finished servicing a request.
  - *Busy-wait termination:* In a broadcast system where access to I/O request and reply queues are implemented by software, rather than by hardware, IOP processes must busy-wait for access to queues.
  - *Hardware errors:* Processor, switch, memory, power failure, soft reset.

**CPU Interrupts.** In a broadcast system implementing *priority preemption*, whenever a process is moved to a ready queue, its priority is broadcast to all processors, as explained in Section C.2.2. If its priority is higher than that of a running process, hardware at the latter CPU generates an interrupt. If the process has this interrupt enabled, it goes to sleep — placing itself on a ready queue — and loads the process of higher priority from its ready queue (Figure 25).

In addition, if the process is moved from a semaphore wait-queue to the ready queue, it now holds privileged access to the data controlled by the semaphore (it holds the 'lock'), so its priority should be raised temporarily. An easy way to implement this is to devote the upper bits of the priority word to this purpose. They are normally set to their minimum value, but in this case are temporarily raised to an appropriate level.

A multiprogrammed system will probably implement *time quanta* in order to maintain sufficiently low turnaround or response times for short or interactive programs, respectively. In this case, when a process is loaded into a CPU to run, a CPU timer is loaded with the process's time quantum. If the timer exhausts its count before the process terminates or blocks for I/O, it generates an interrupt, and if the interrupt is enabled, the process places itself on a ready queue and loads the highest priority process from the queue. The latter may actually be the original process, in which case it need not save and reload its state. If an interrupt is not enabled at the time it is generated, it is

subsequently serviced when it is enabled.

In a broadcast system that implements *priority preemption as well as time quanta*, the preemption activity described above is also generated. That is, when the process that has exhausted its time slice puts itself on a ready queue, its priority is broadcast to all CPUs. Consequently, a lower priority process running on another CPU may be preempted, and the two processes will end up simply interchanging CPUs. An improvement in this case is for the lower priority process to signal the higher when the higher is putting itself on the queue, thereby aborting the enqueuing and allowing the higher to continue running where it is.

Recall that time quanta are of value for insuring reasonable turnaround and response times for the users of a multiprogrammed or interactive system, respectively. Consequently, if the system is, at the other extreme, executing a single program with many processes, the priorities of the processes should be sufficient to manage the timesharing of the CPUs among the processes. In this case, quantum timing would generate needless, wasteful process switching, and should not be used.

As we have also seen, in a broadcast system, a busy-waiting process is notified via broadcast and an interrupt that it can now access the data (Section B.2.2).

Finally, routine maintenance interrupts, such as time-of-day interrupts, will cause a temporary switch to the operating system, and then back again to the user process. Whereas hardware errors may cause the processor to retry an action or to halt. The latter would not be disabled when a lock is locked.

**IOP Interrupts.** An I/O device that has just become ready or has just finished servicing a request will interrupt its IOP. These interrupts will differ in priority, the more urgent interrupts being given higher priority. For example, disks will have higher priority than terminals, and unbuffered I/O devices will take precedence over buffered devices.

Regarding *busy-wait*, if the hardware does not implement sleep-wait queuing, an IOP process will have to busy wait on system process and I/O queues when seeking access. Unlike a CPU, however, an IOP should be able to manage I/O without process-switching, thereby making I/O management more efficient and ruling out the arbitrarily long wait that is possible in process switching. However, *hardware sleep-wait queuing* does away with the need for busy waiting to gain access to the system queues, making I/O even more efficient, as discussed in Section C.1.

As noted many times, process-switching interrupts are disabled at strategic moments during synchronization activity, typically while a process has access to write-shared data — atoms. The most general reason is to limit the time that the data is locked and therefore unavailable to other processes, in particular, to preclude arbitrarily long waiting. The same argument also applies to disabling process-switching interrupts while busy waiting under the FIFO scheme (Appendix 4). Other reasons may apply in specific situations.

From the preceding enumeration of CPU interrupts, we can at last get some feel for the system effects of a CPU disabling process-switching interrupts. The productivity of the system will suffer to the extent that too many CPUs disable switching too long, leaving an insufficient number to service the *preemption* requests — to run higher priority processes — as rapidly as needed. Response time of a multiprogrammed system will suffer to the extent that too many CPUs disable switching too long, leaving an insufficient number to service the *quantum timer* requests as rapidly as needed. (The terminal I/O requests are handled by IOPs, not CPUs.) Finally, maintenance tasks will be neglected if their interrupts are disabled for too long. But the specific effects on performance can only be determined through further analysis, followed by stochastic modeling and simulation.

# Figures

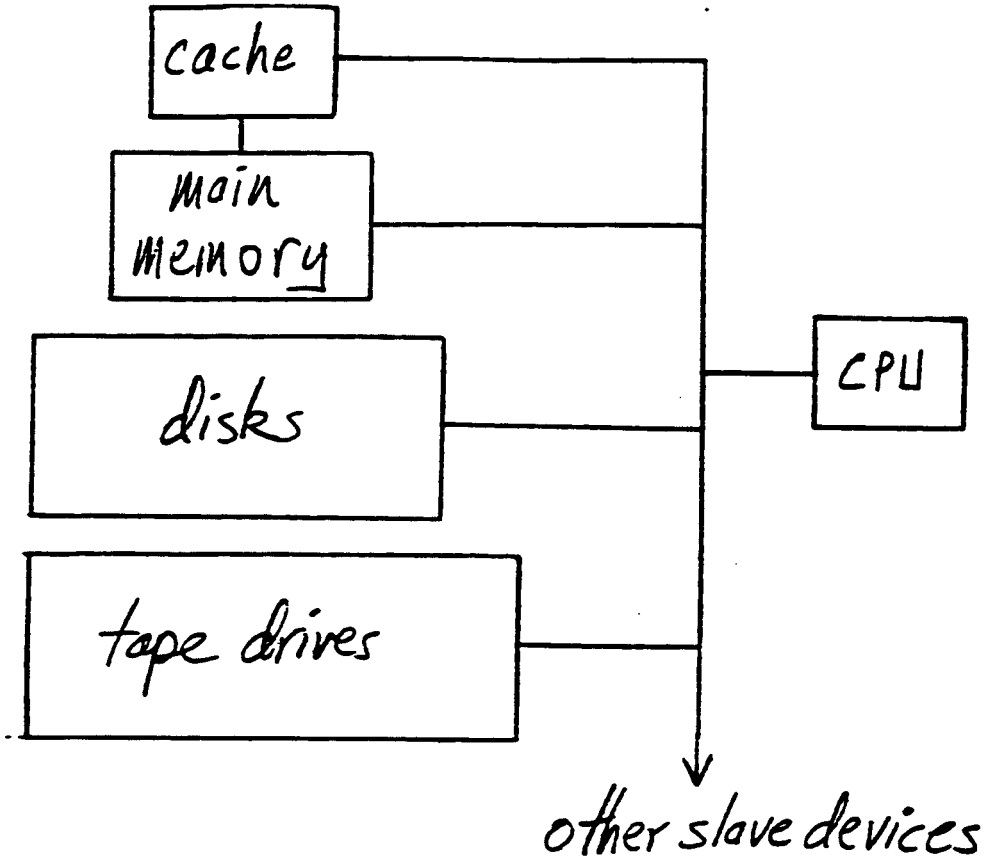



Figure 1. Single-processor system (one CPU, no IOPs);
, cache serves as high-speed component of memory hierarchy.

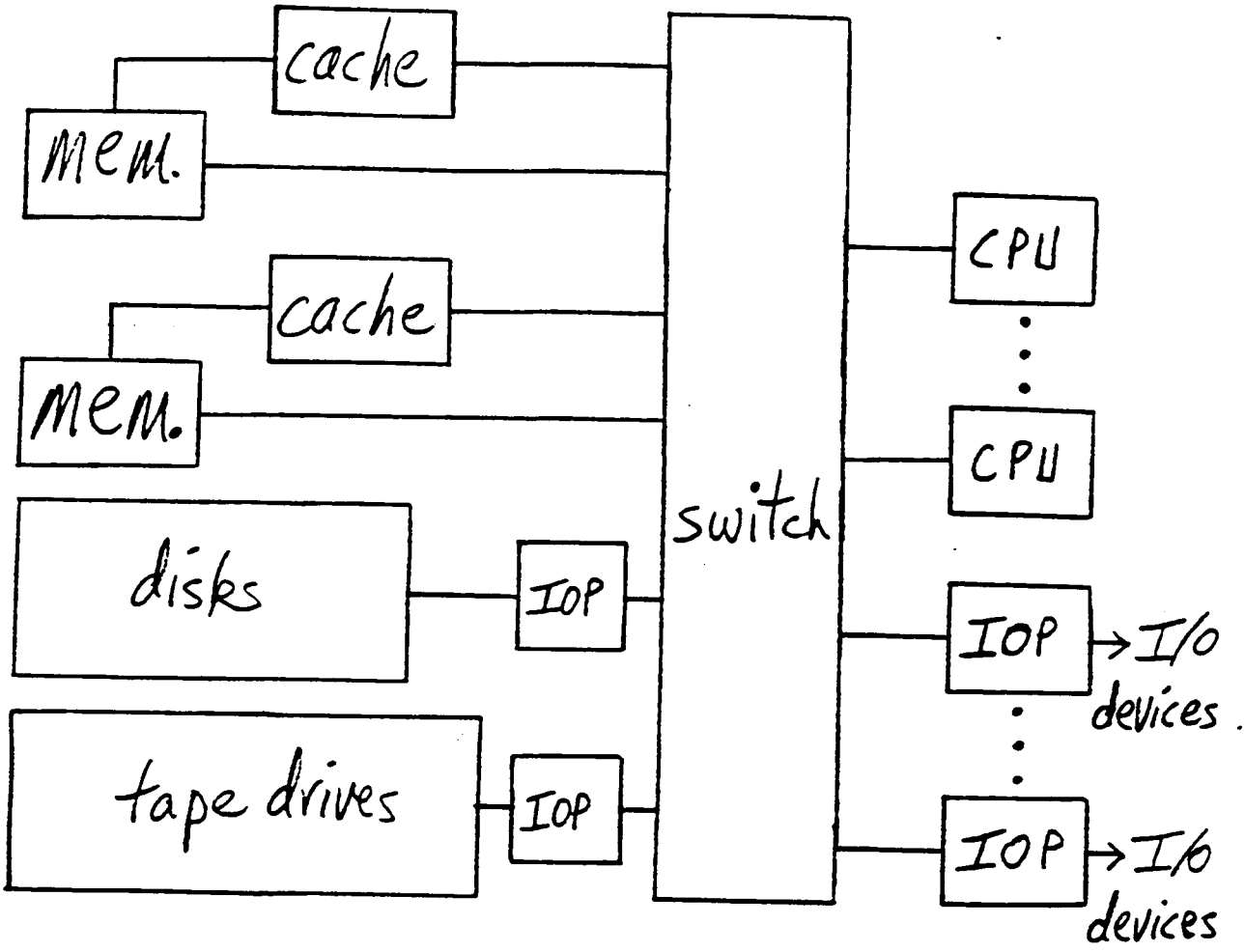



Figure 2. Multiprocessor system where cache serves only as high-speed component of memory hierarchy.

Note: the switch is generic, able to connect any two devices together.

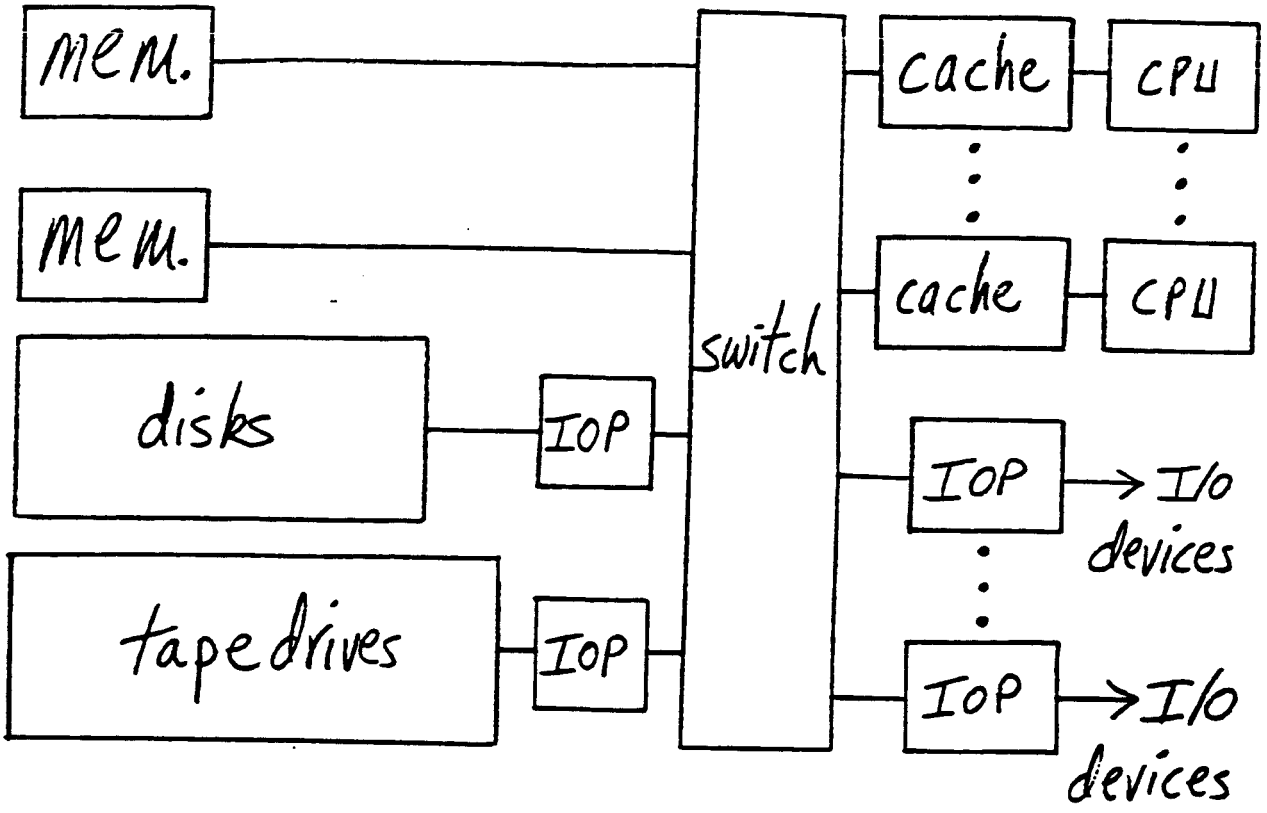



Figure 3. Multiprocessor system where caches also serve as local memories.

Note: the switch is generic, able to connect any two devices together.

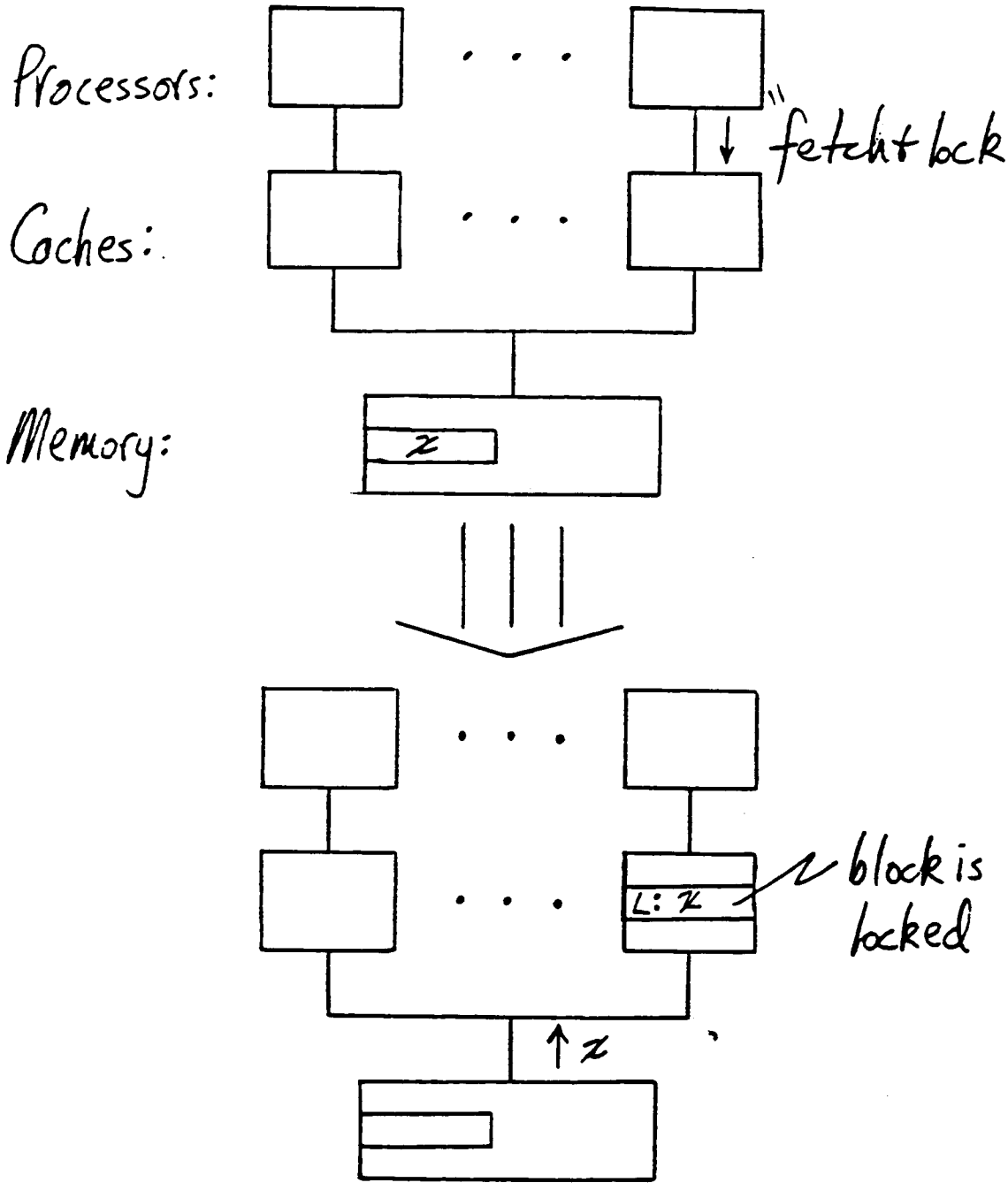



Figure 4. First block x of an object is fetched to cache and locked.

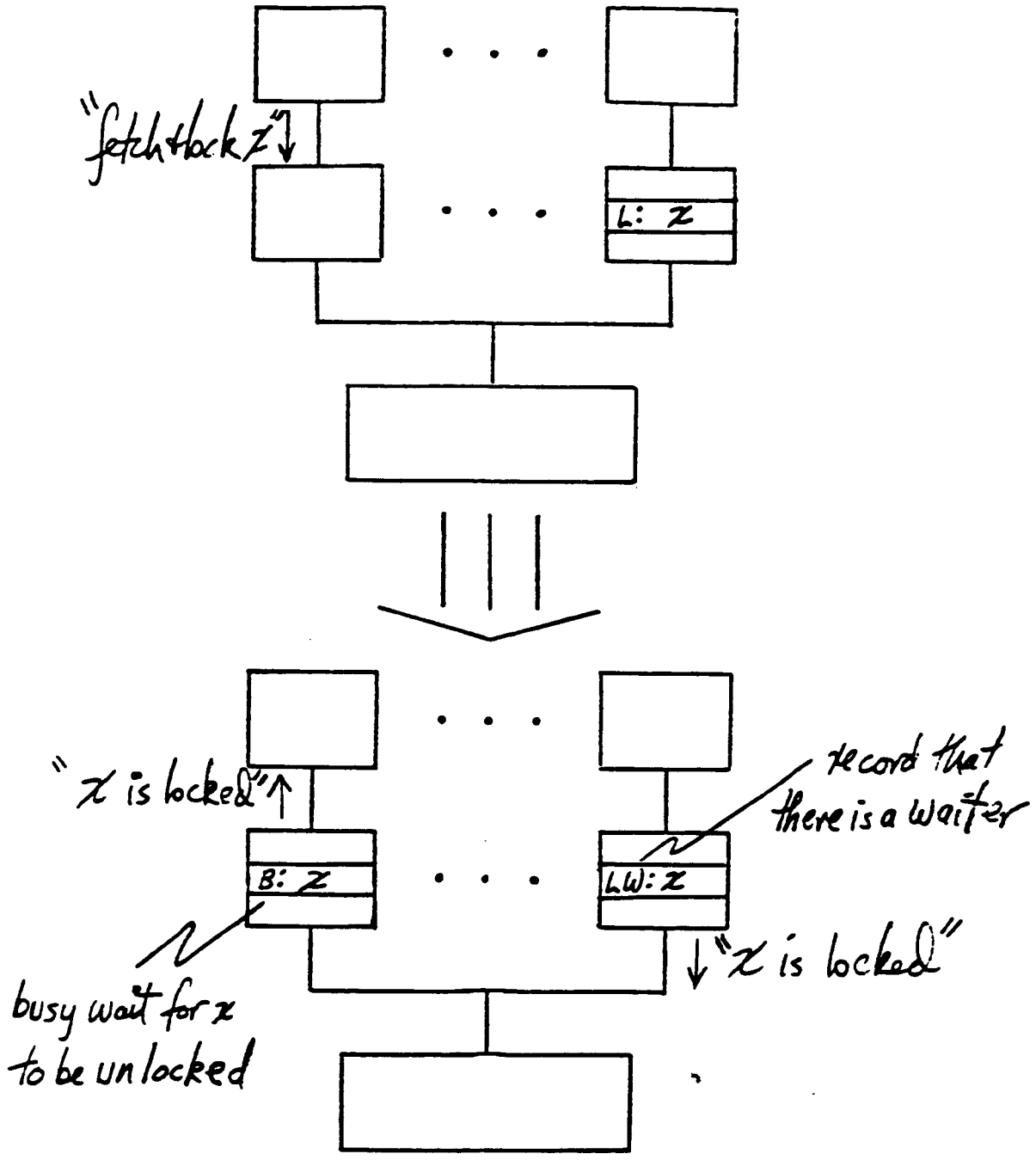



Figure 5. Cache holding locked block reports that Requester cache waits for block to be unlocked.

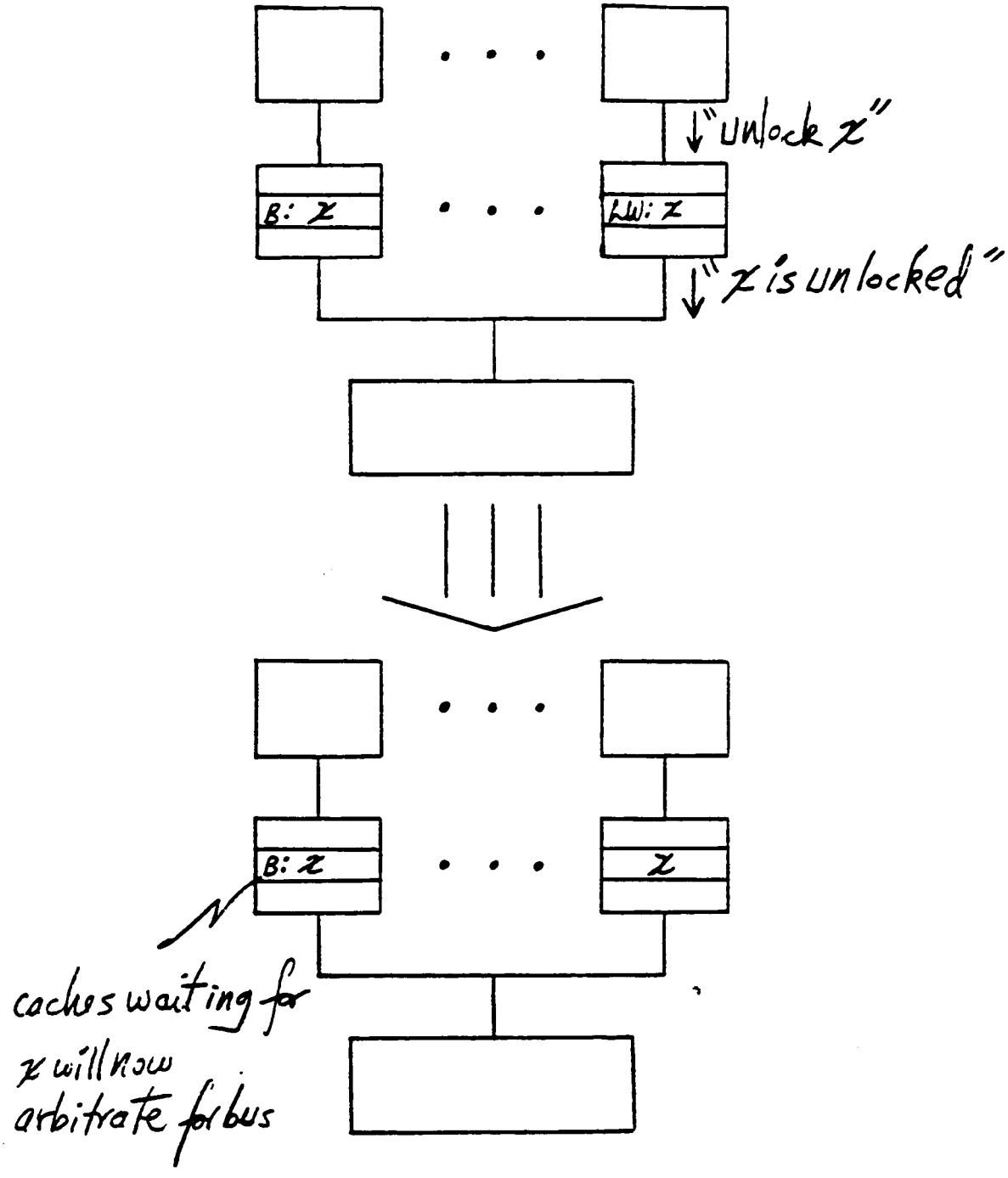



Figure 6. When processor unlocks block, it broadcasts this if there is a waiter. Waiters then arbitrate.

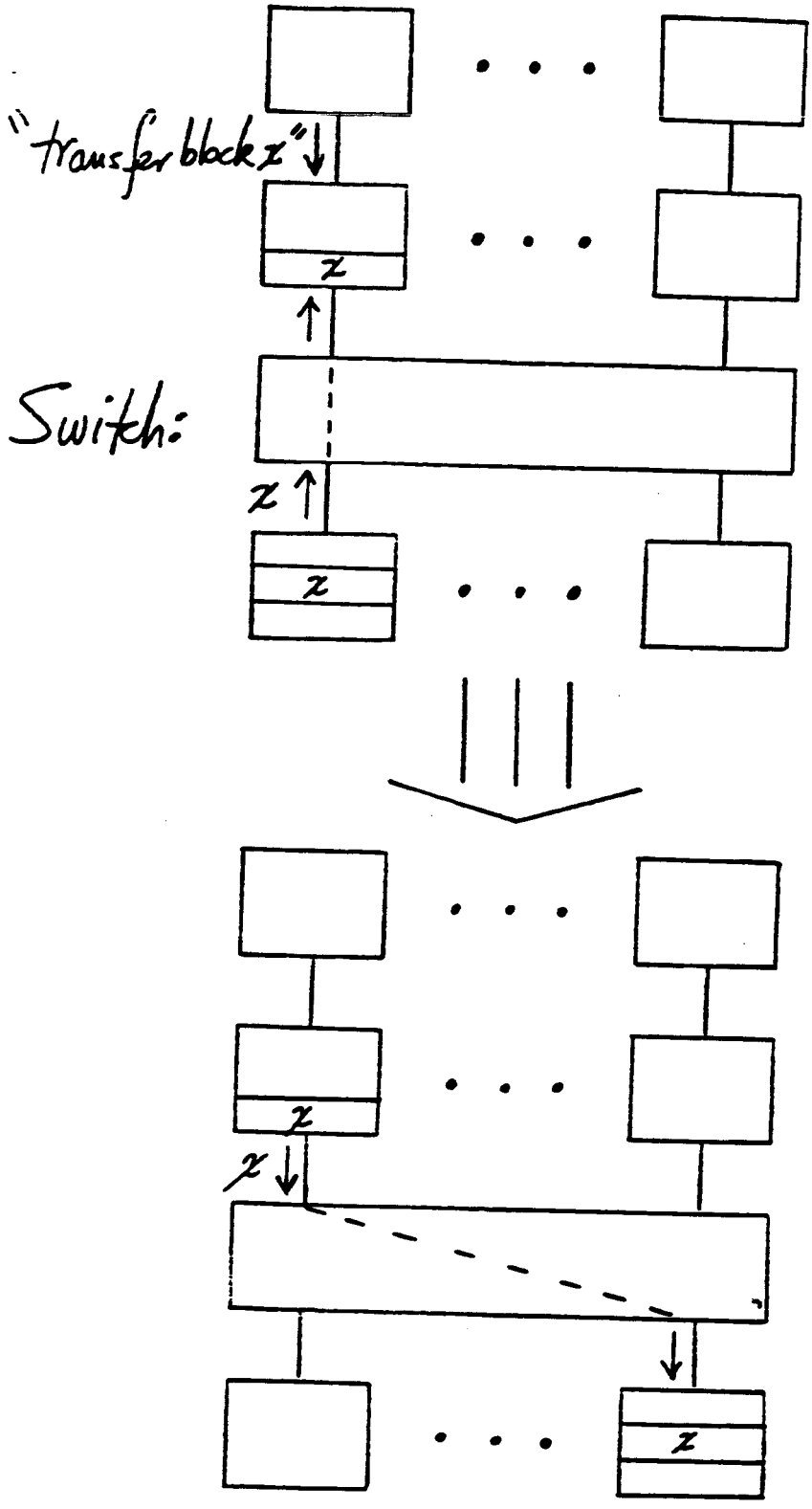



Figure 7. Cache-mediated memory-to-memory transfer.

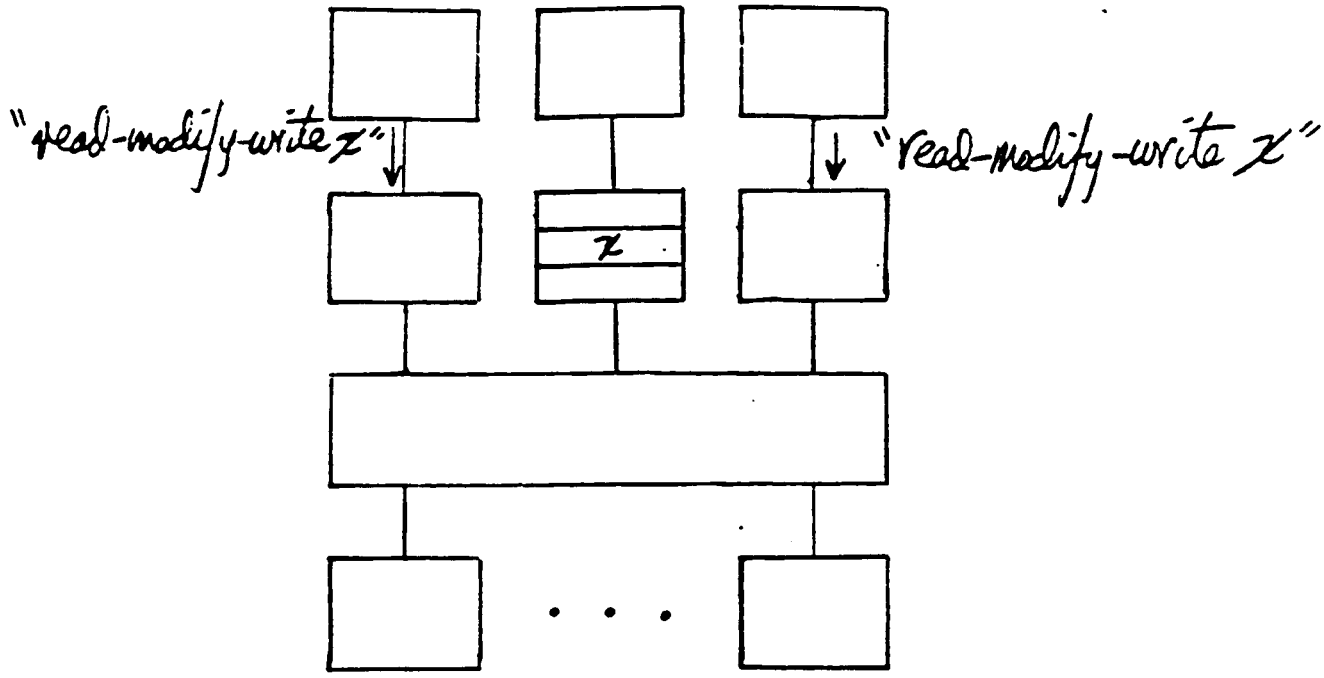



Figure 8. Hardware serializes conflicting access requests — hard atoms only.

Examples of atomic read-modify-writes include test-and-set and atomic add.

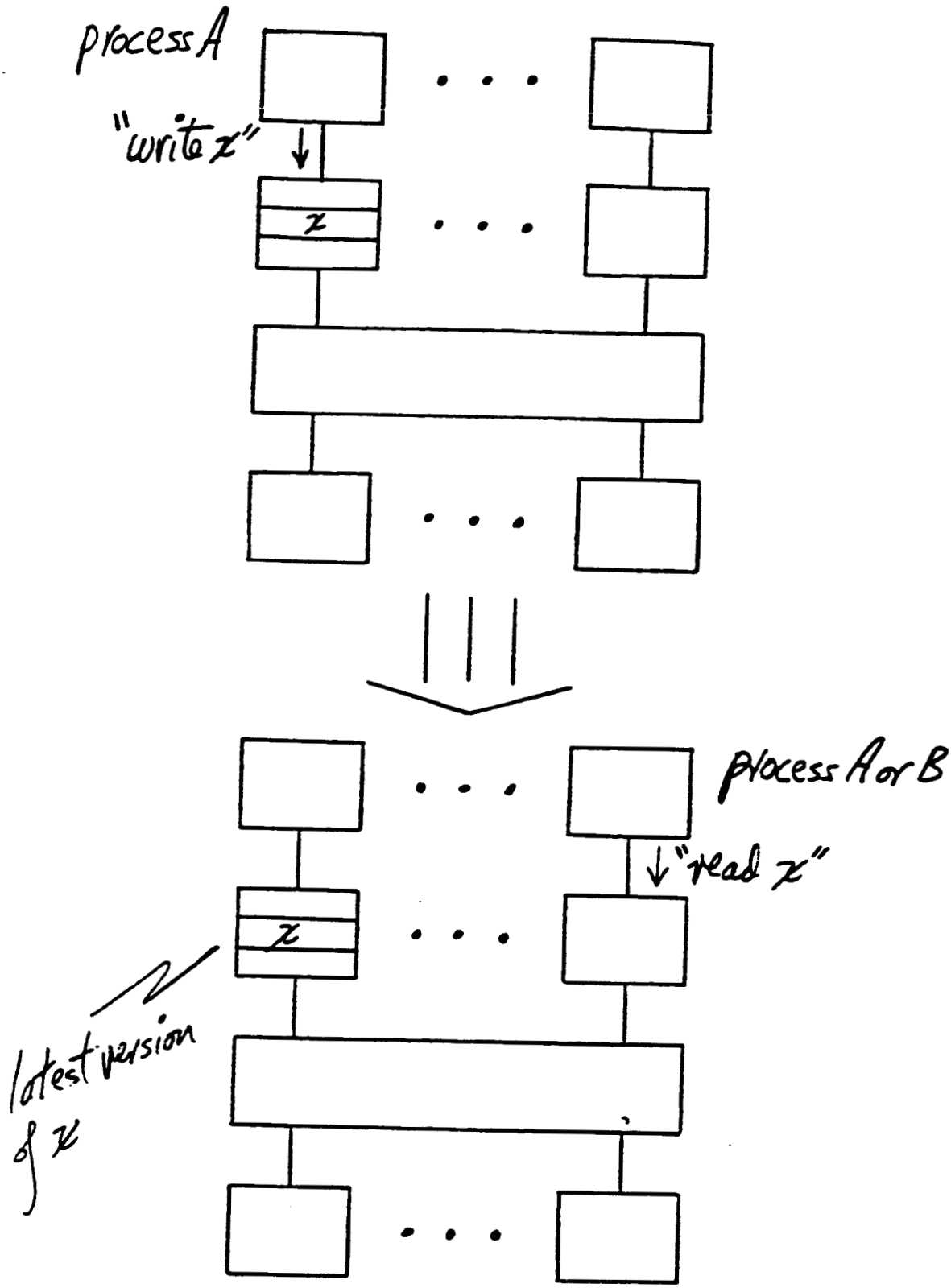



Figure 9. Hardware provides latest version of requested data — all objects.

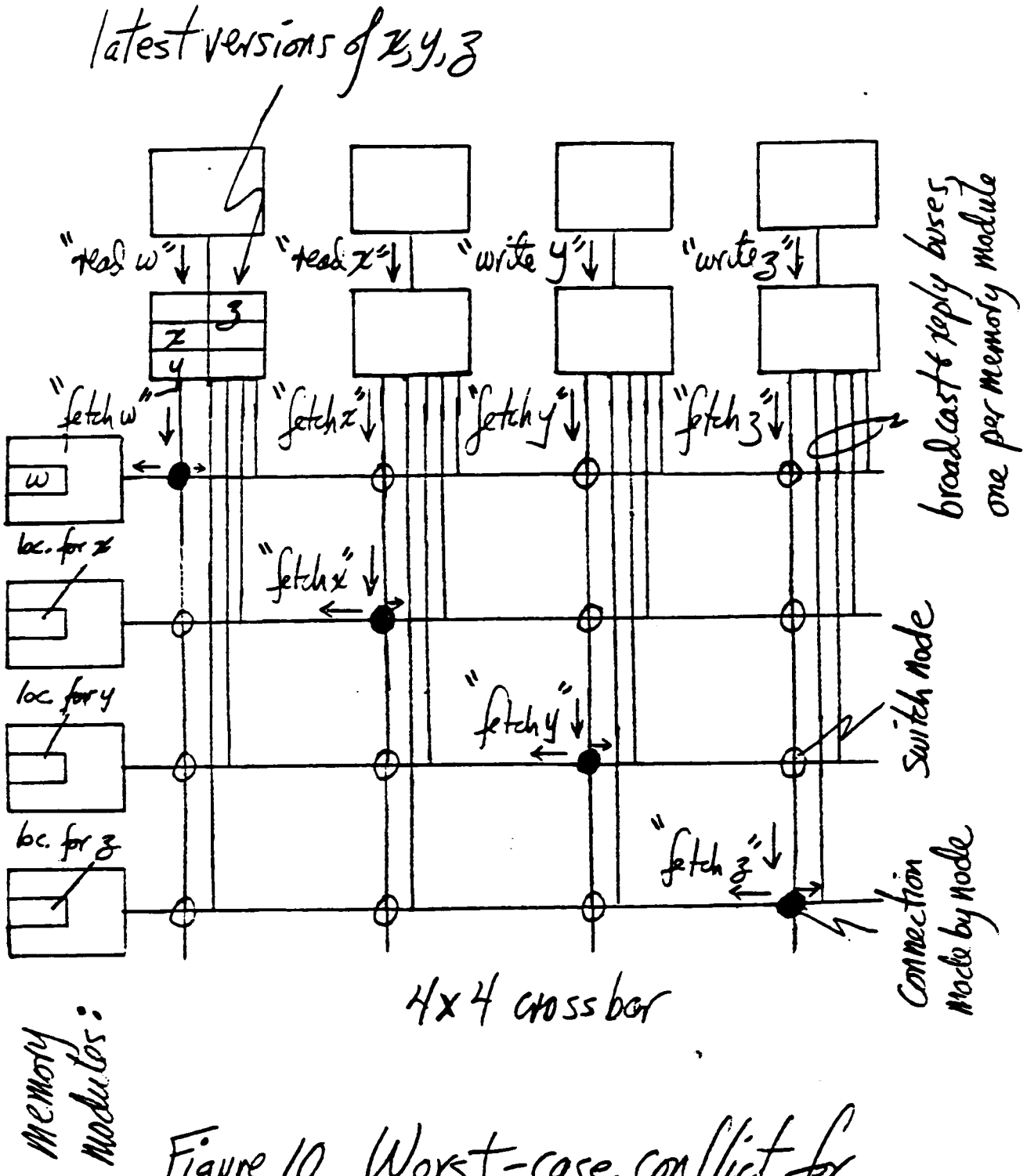
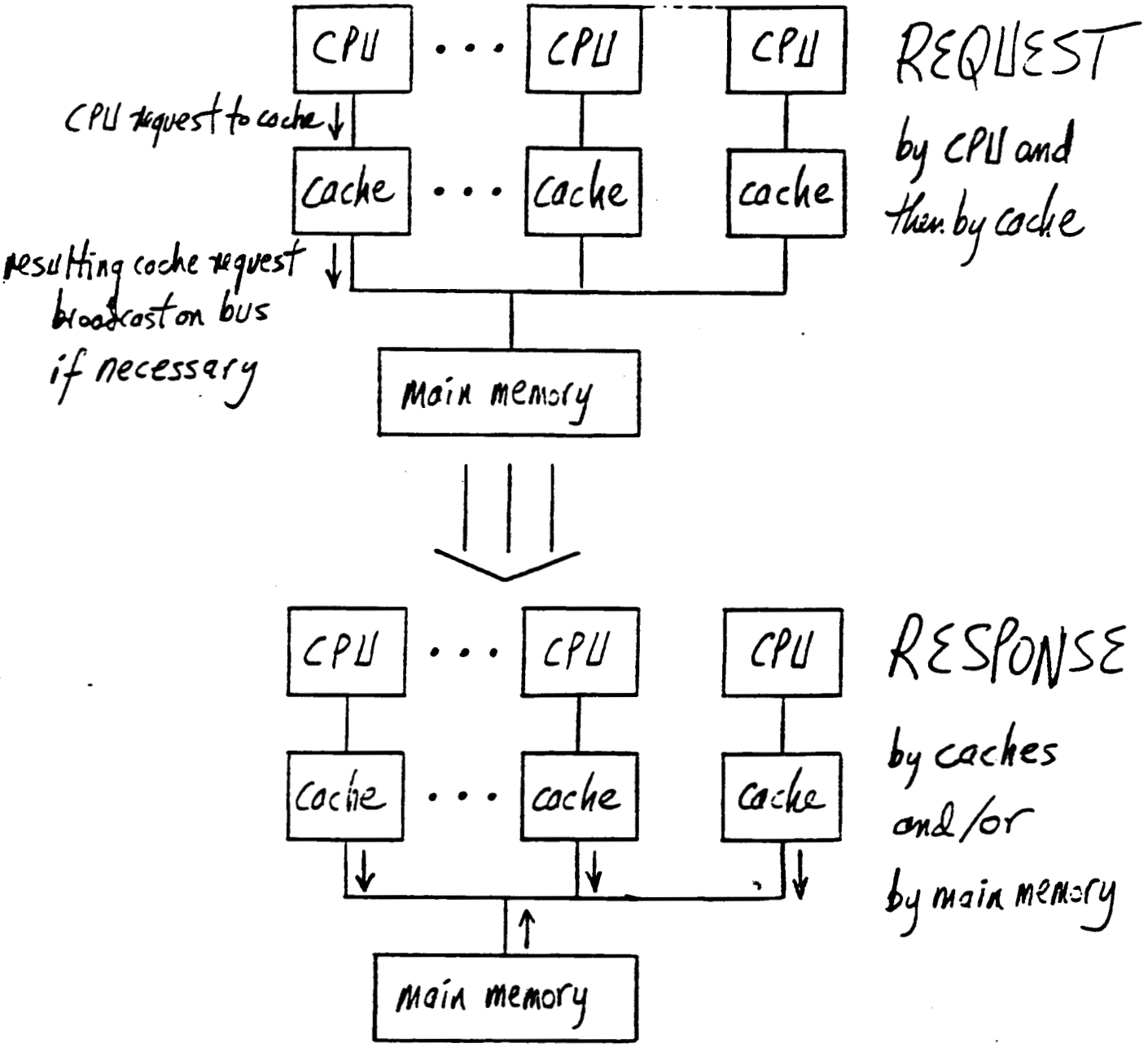



Figure 10. Worst-case conflict for n-way (4-way) broadcast system having one cache per processor.




Template: this template shows the format of the subsequent figures.

## Abbreviations for Subsequent Figures

| | |
|---|---|
| *B* | Busy-wait register is loaded |
| *block* | target block |
| *C* | Clean |
| *D* | Dirty |
| *F* | Fetch |
| *I* | Invalid |
| *L* | Lock |
| *LW* | Lock waiter |
| *R* | Read |
| *S* | Source |
| *U* | Unlock |
| *W* | Write |
| *word* | target word |

## Note for Subsequent Figures

A state indicator inside a cache indicates that the target block is present in that cache and has that state. A blank cache indicates that the block is absent or invalid in that cache.

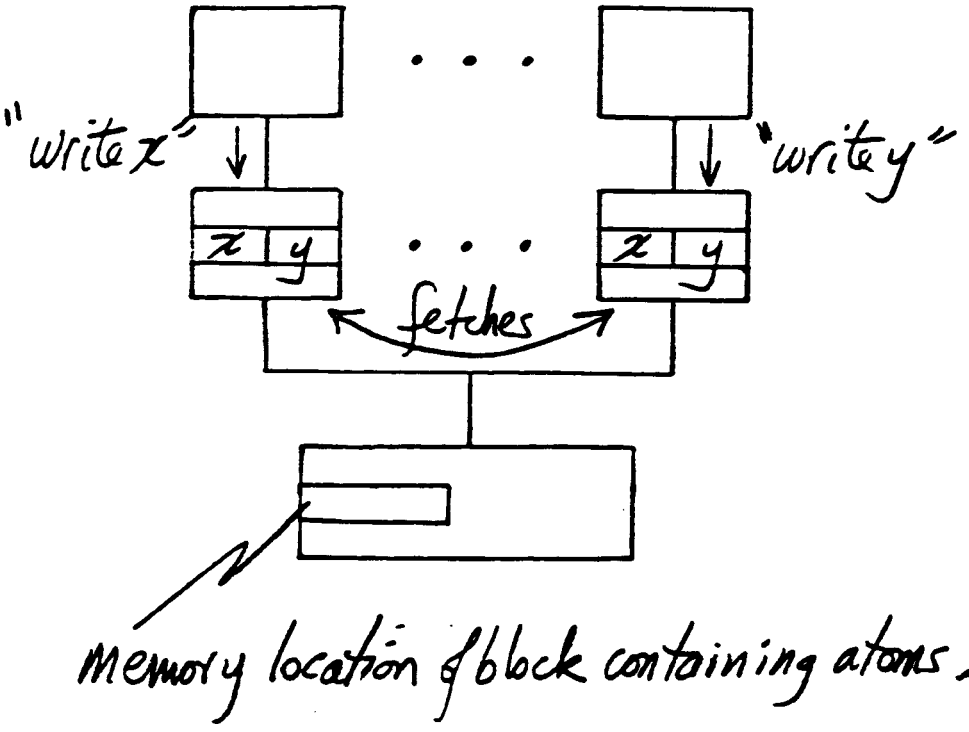



Figure 11. Thrashing of block that contains two atoms.

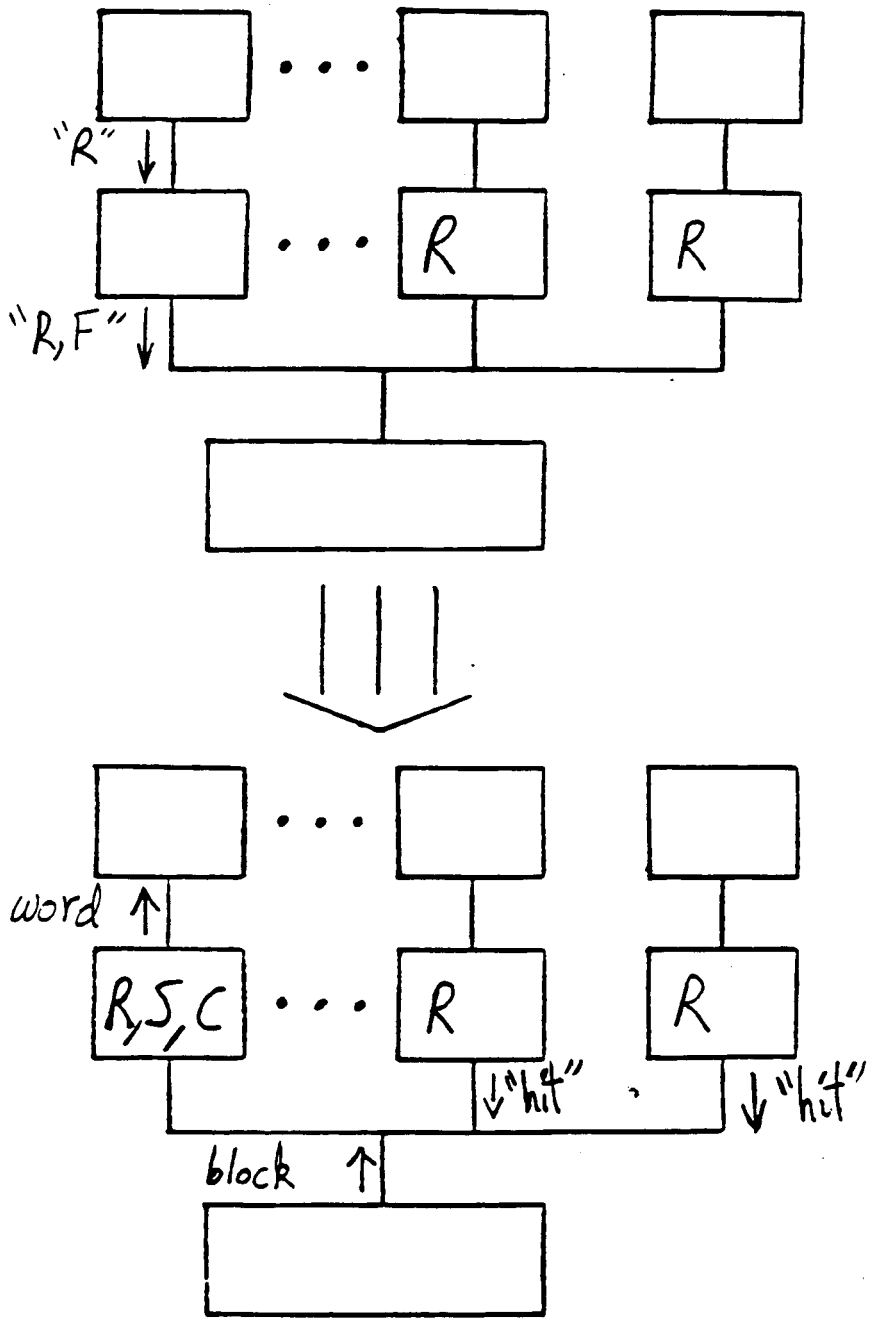



Figure 12. Request for read privilege; no source cache.

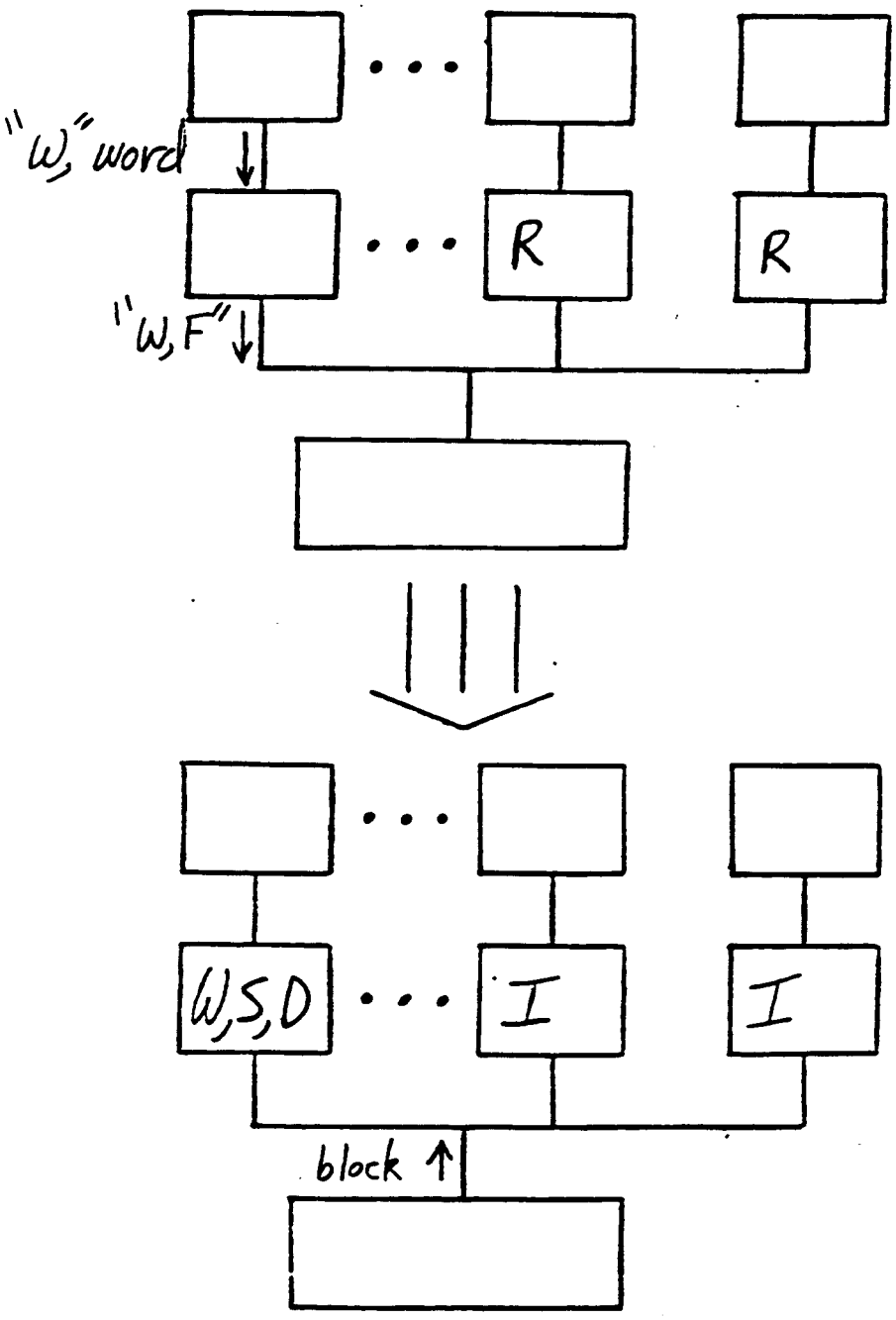



Figure 13. Request for write privilege; no source cache.

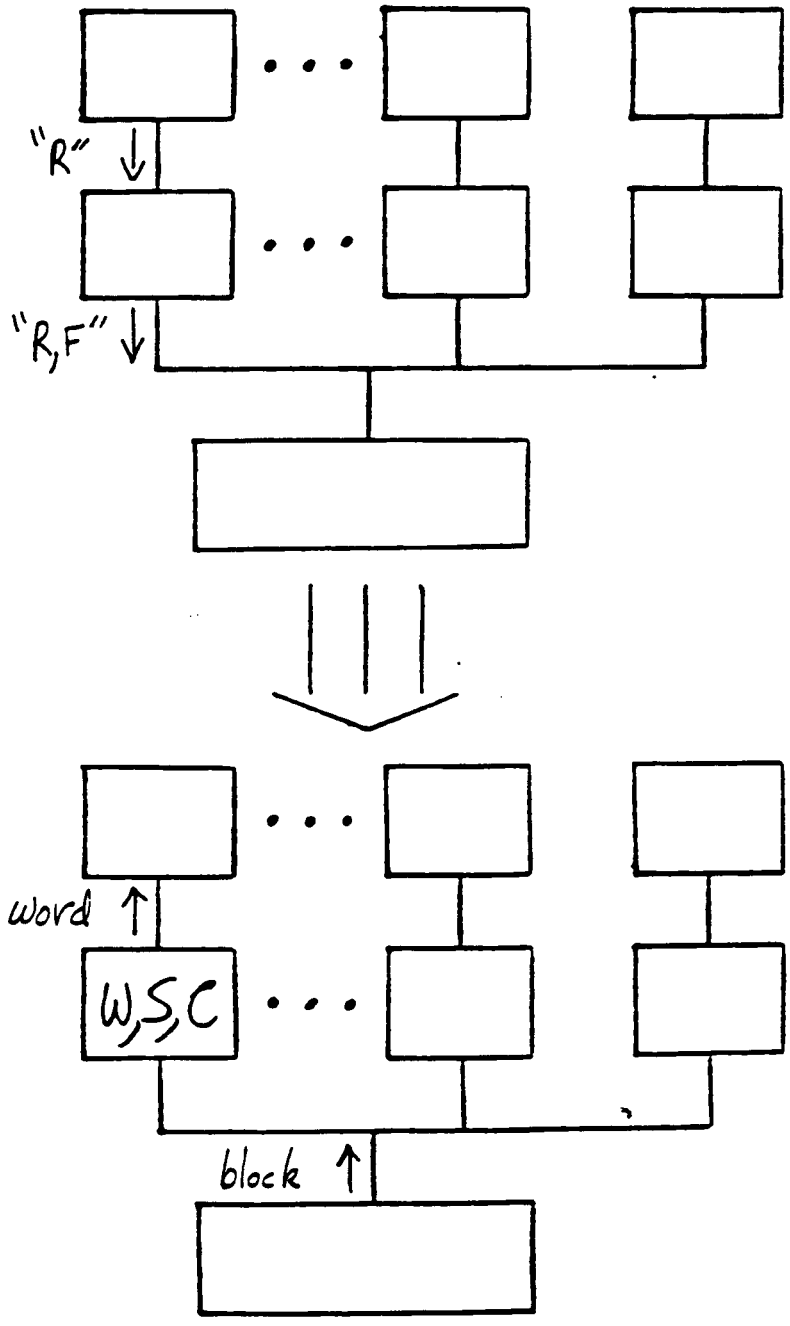



Figure 14. Request for read privilege, block absent (or invalid) in all caches.

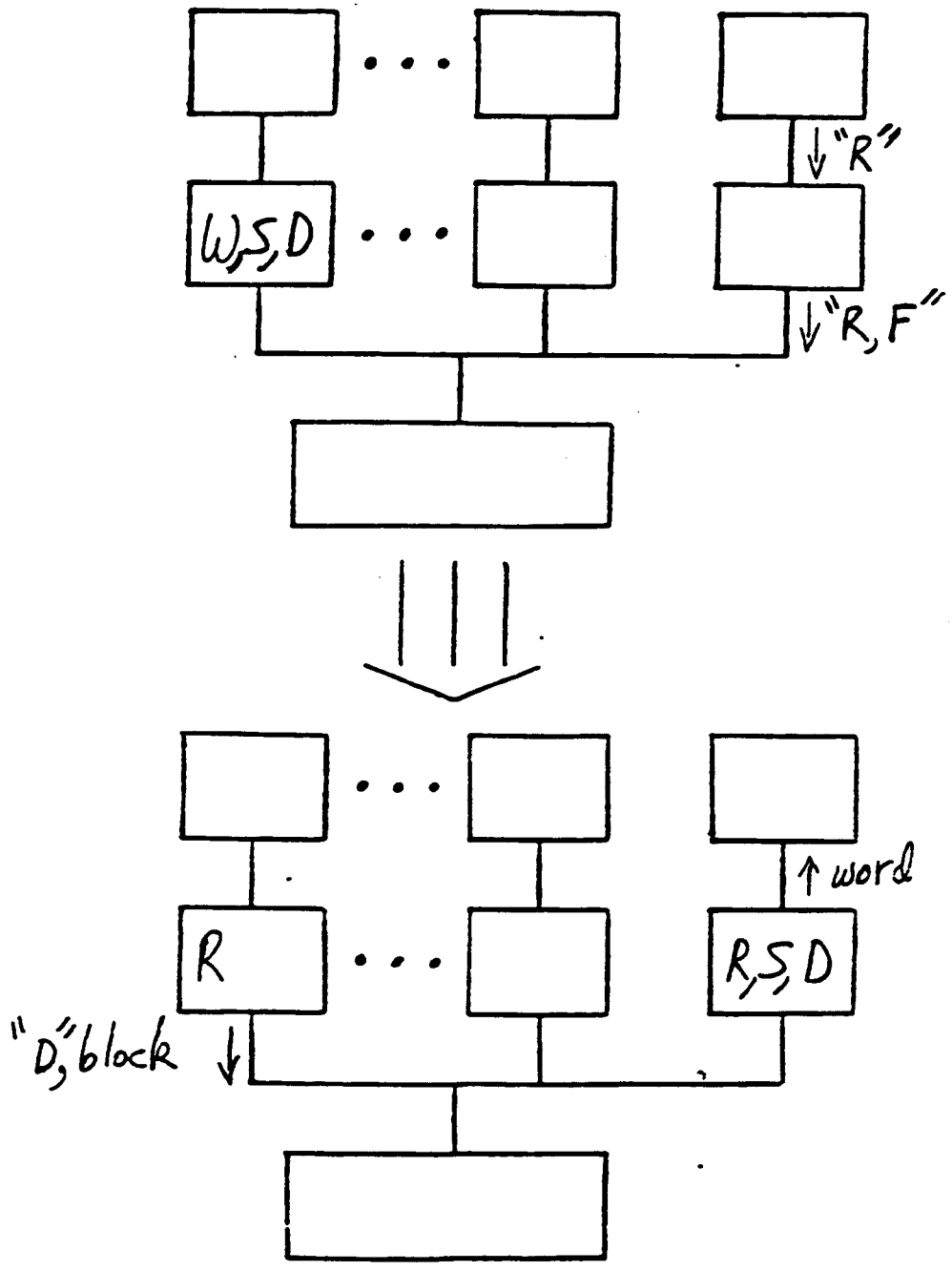



Figure 15. Request for read privilege, source present.

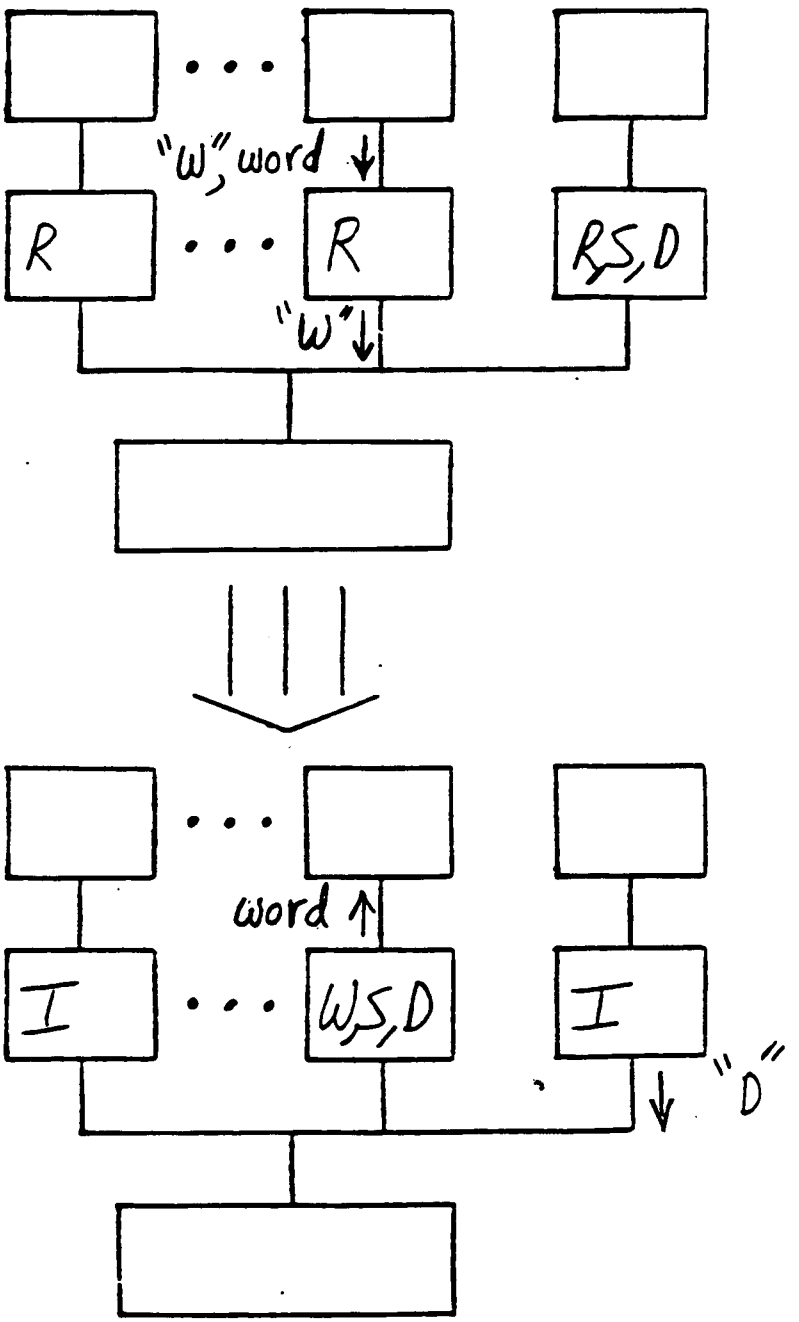



Figure 16. Request for write privilege, but not block itself.

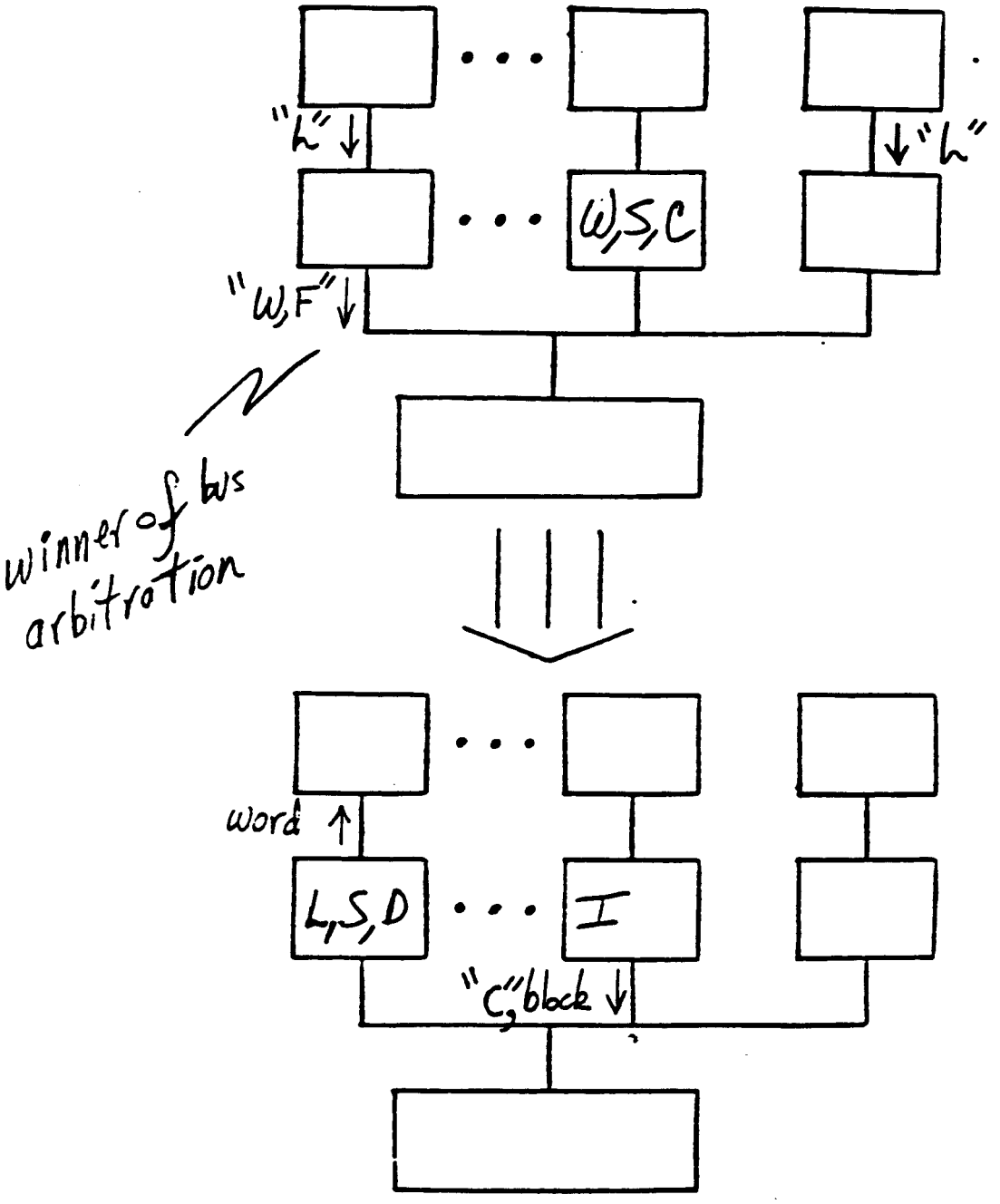



Figure 17. lock request.

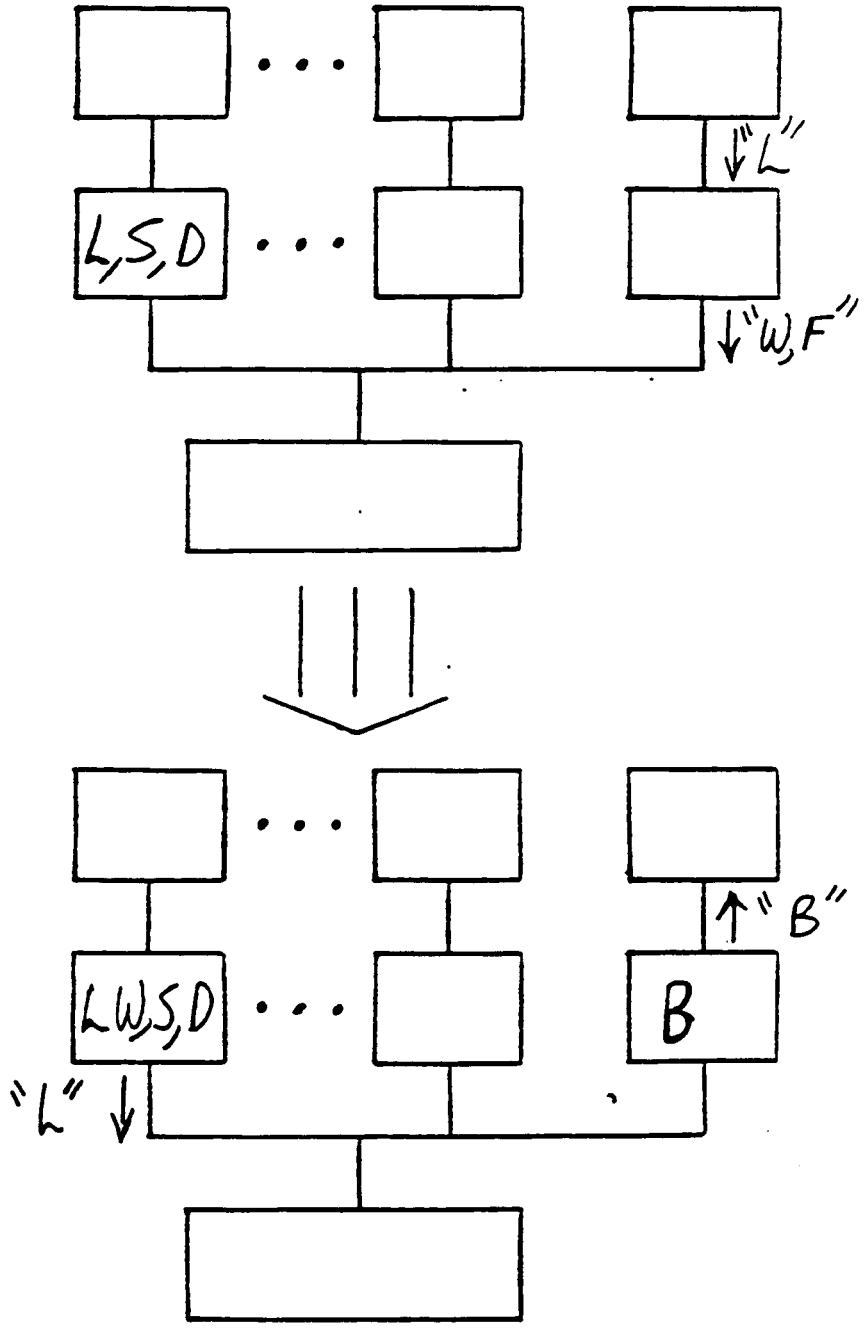



Figure 18. Request for locked block.

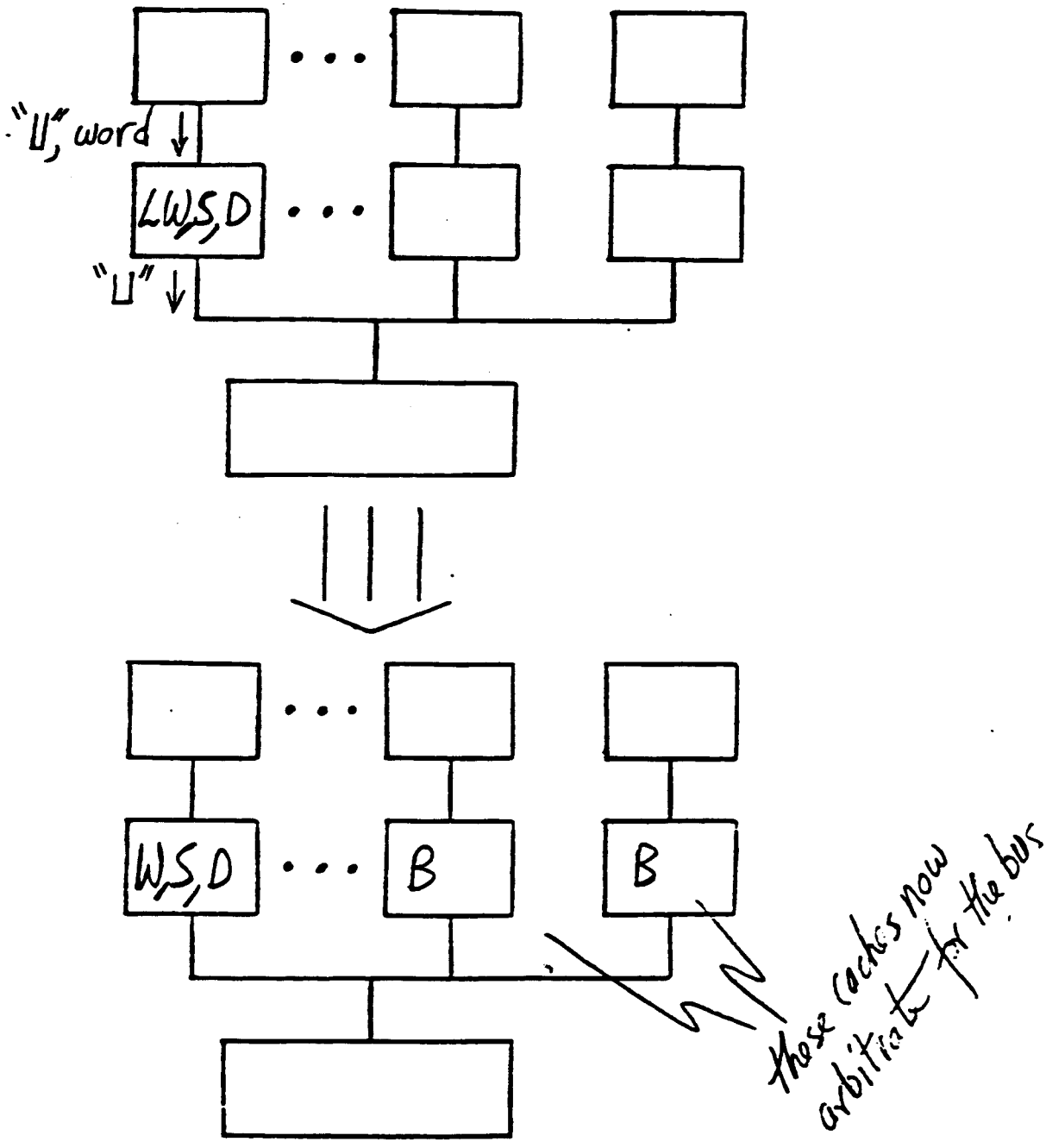



Figure 19. Unlocking a block on which other processors are busy waiting.

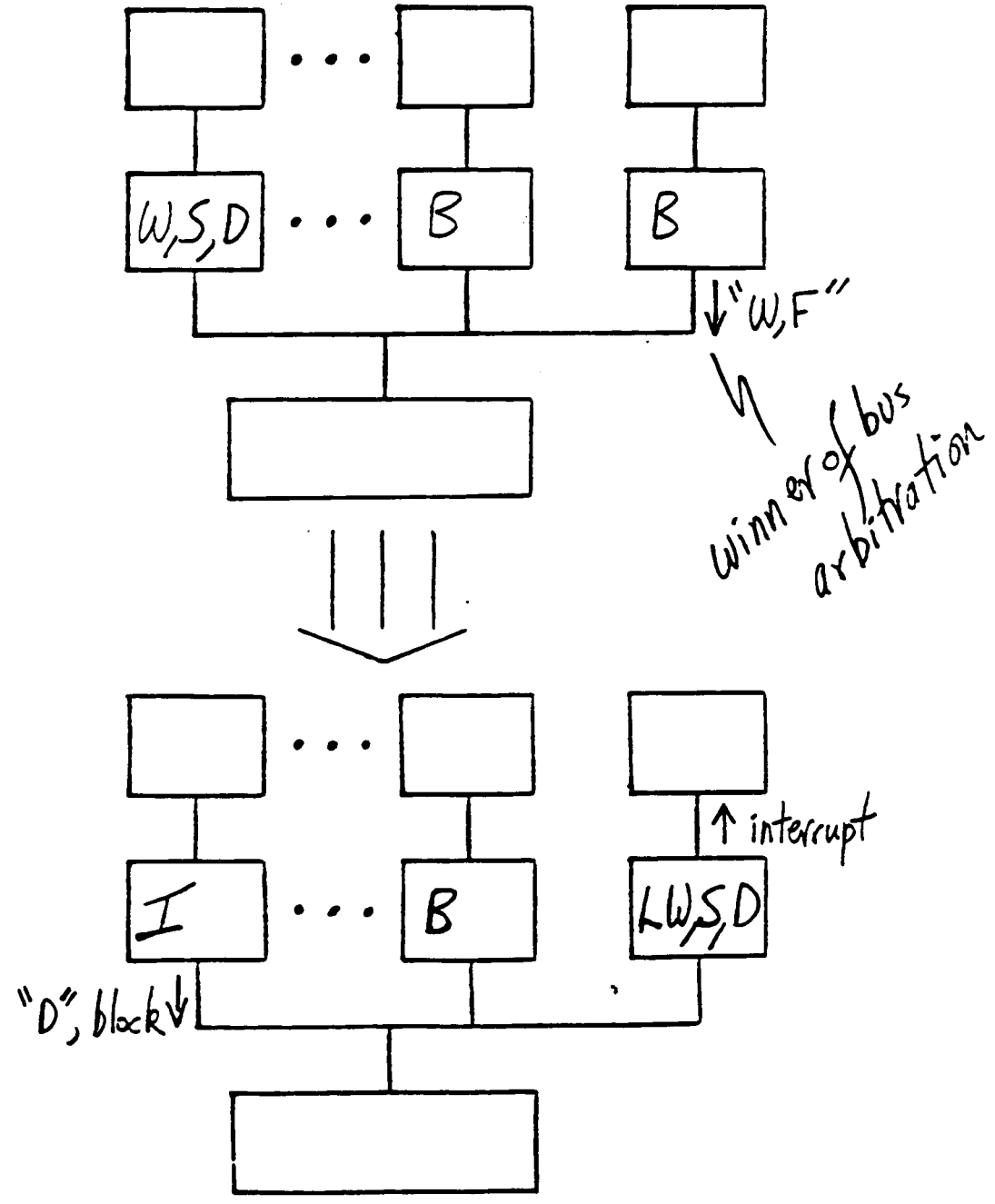



Figure 20. Busy-wait culmination.

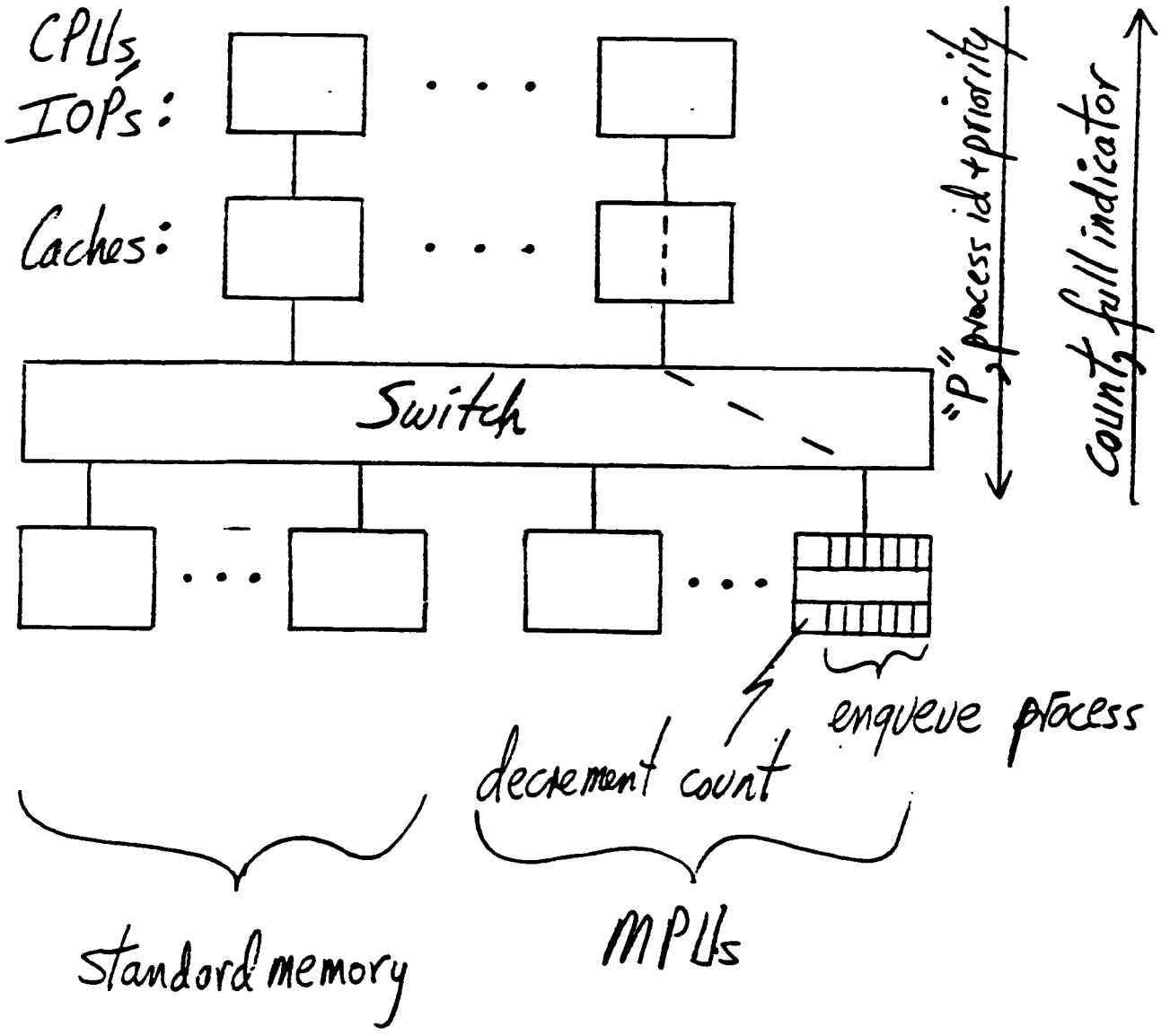



Figure 21. Enqueuing, or P, operation.

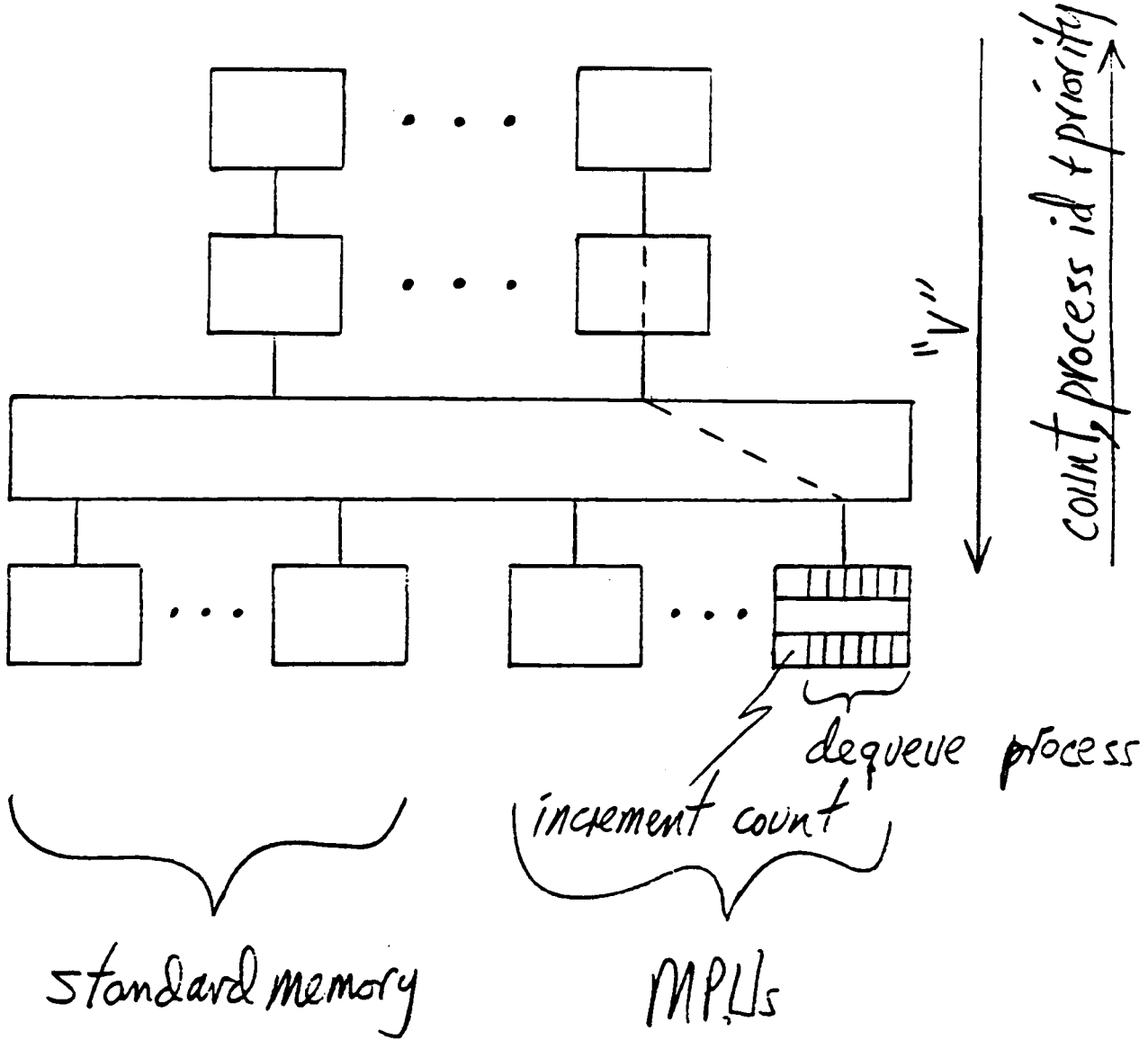



Figure 22. Dequeuing, or V, operation.

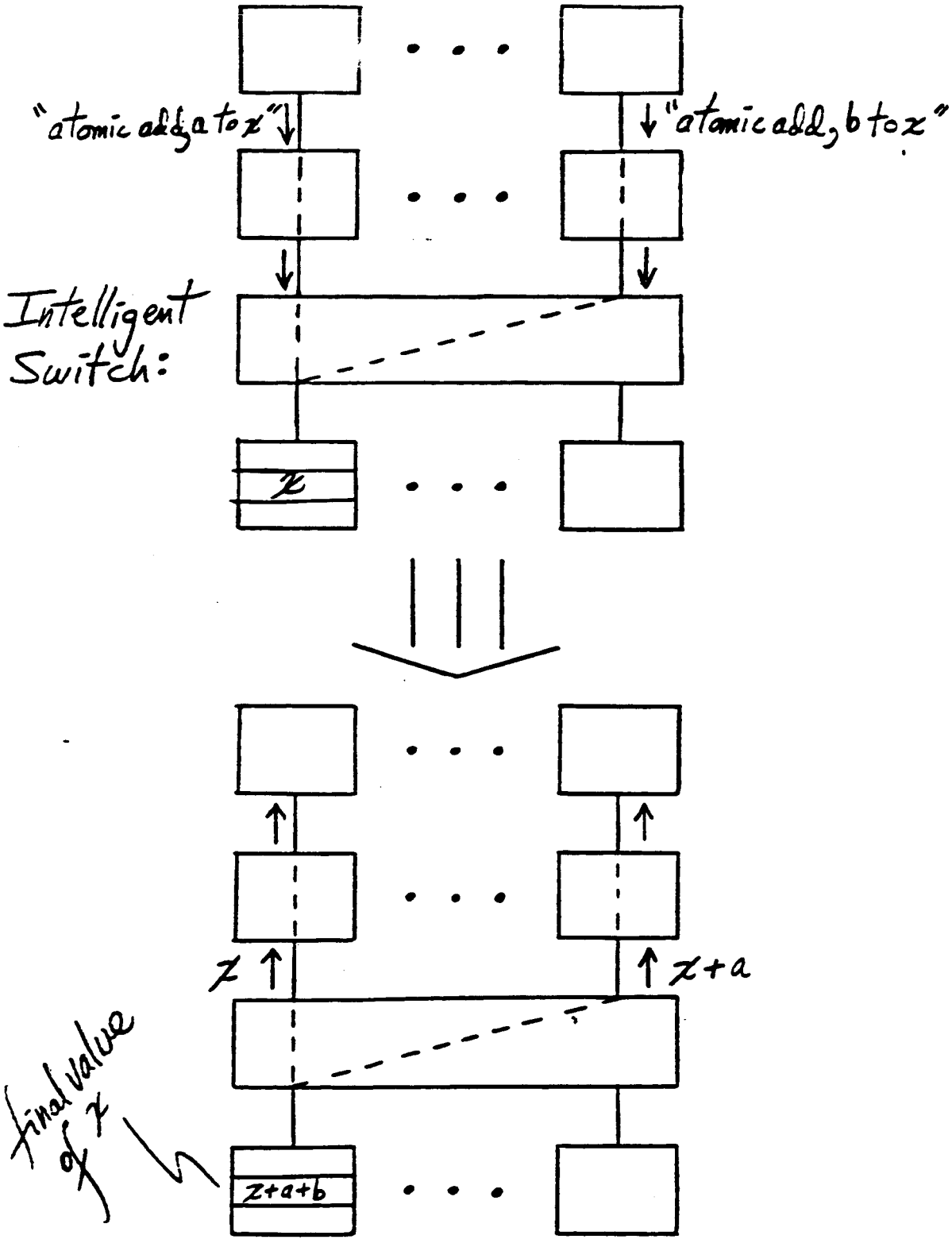



Figure 23. Switch executes atomic operations, serializing conflicting requests.

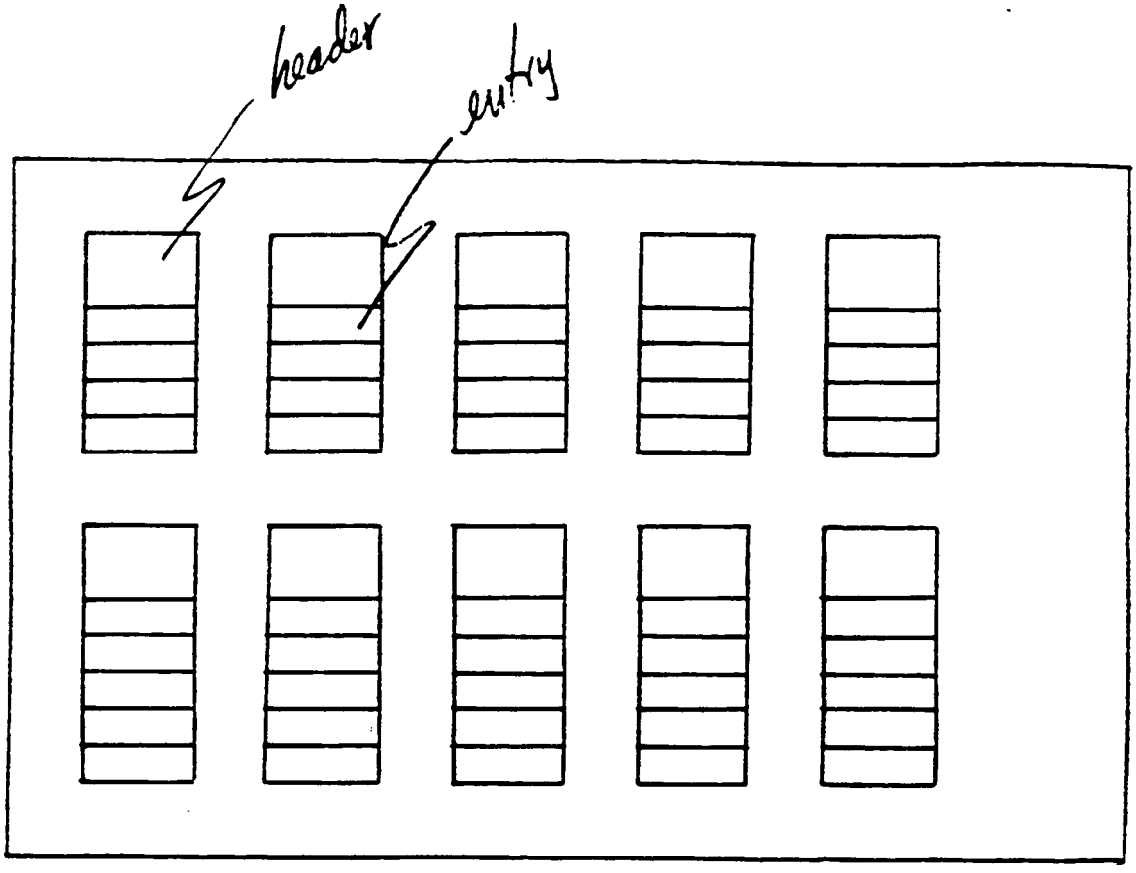



Figure 24. Organization of VLSI queue chip.

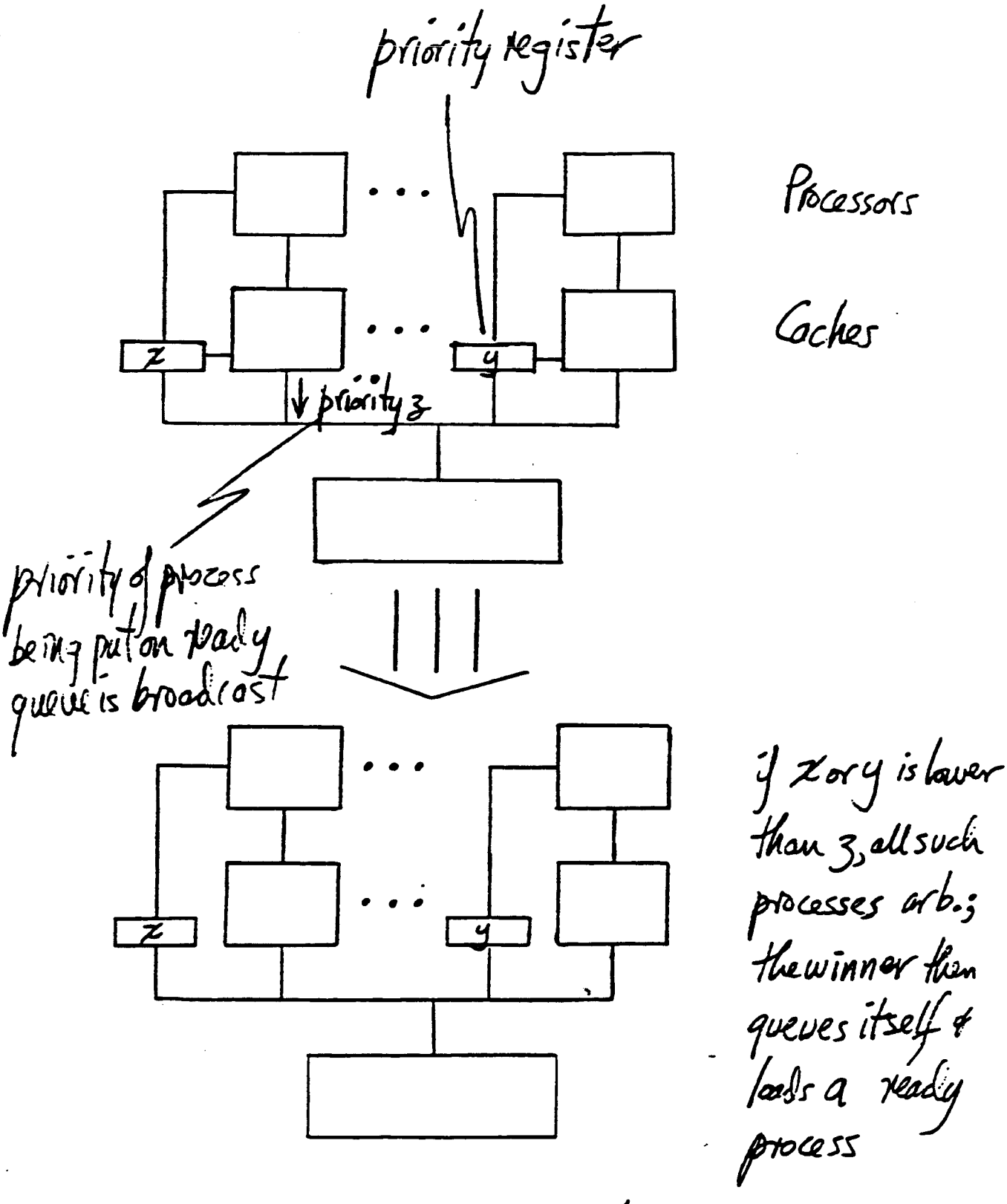



Figure 25. Priority preemption in broadcast system.

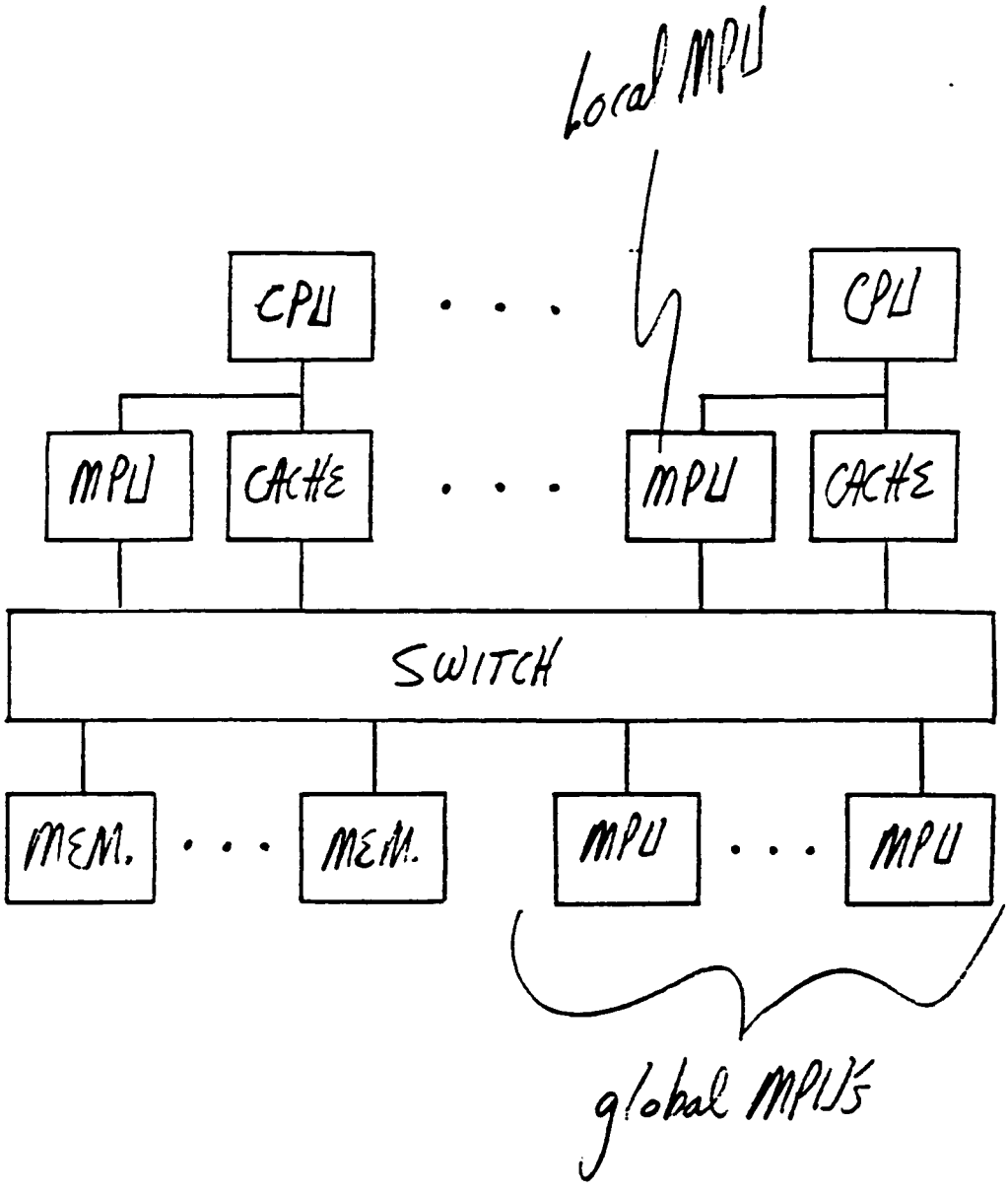



Figure 26. System with local MPUs.

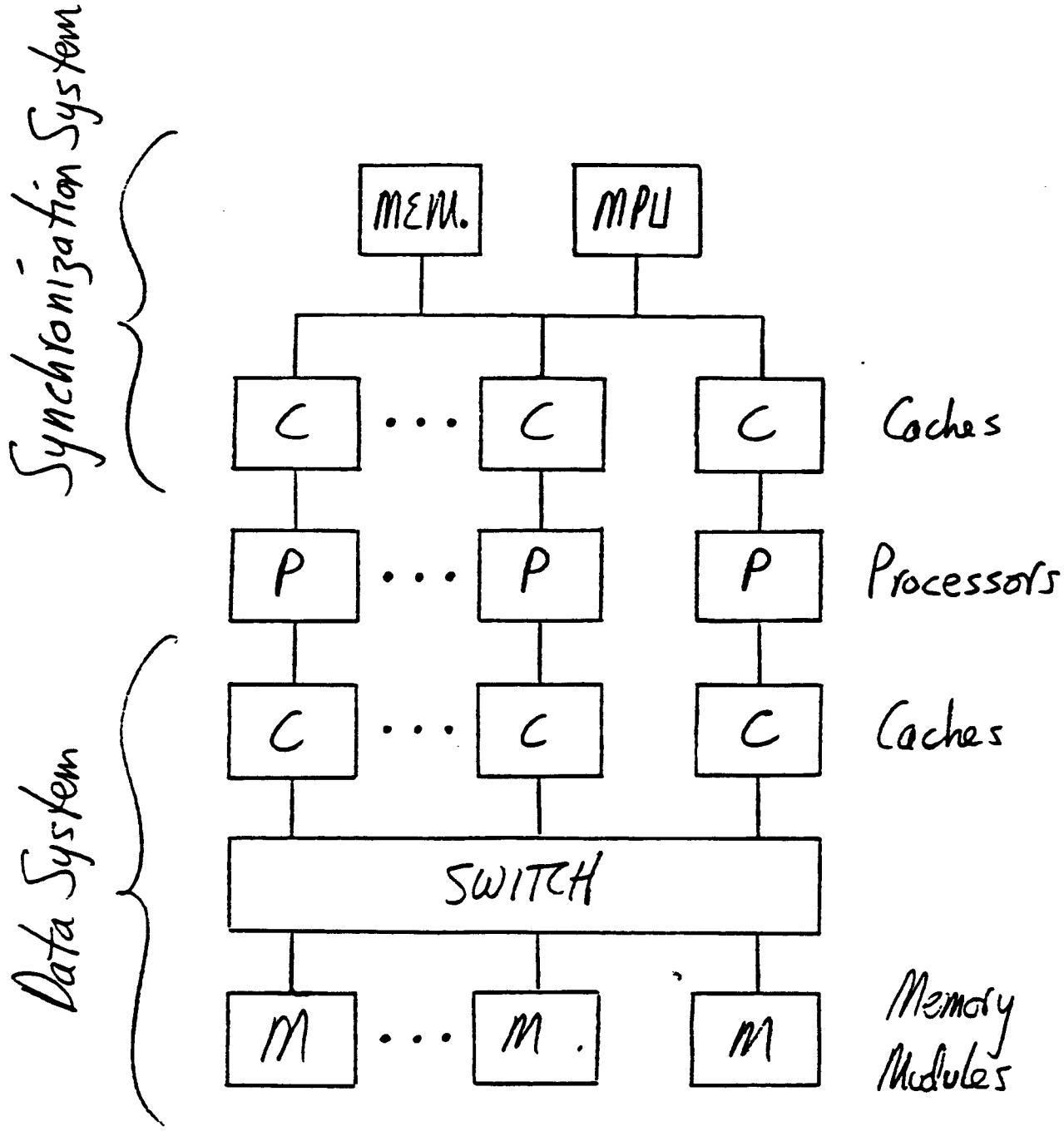



Figure 27. Aquarius split-level architecture.

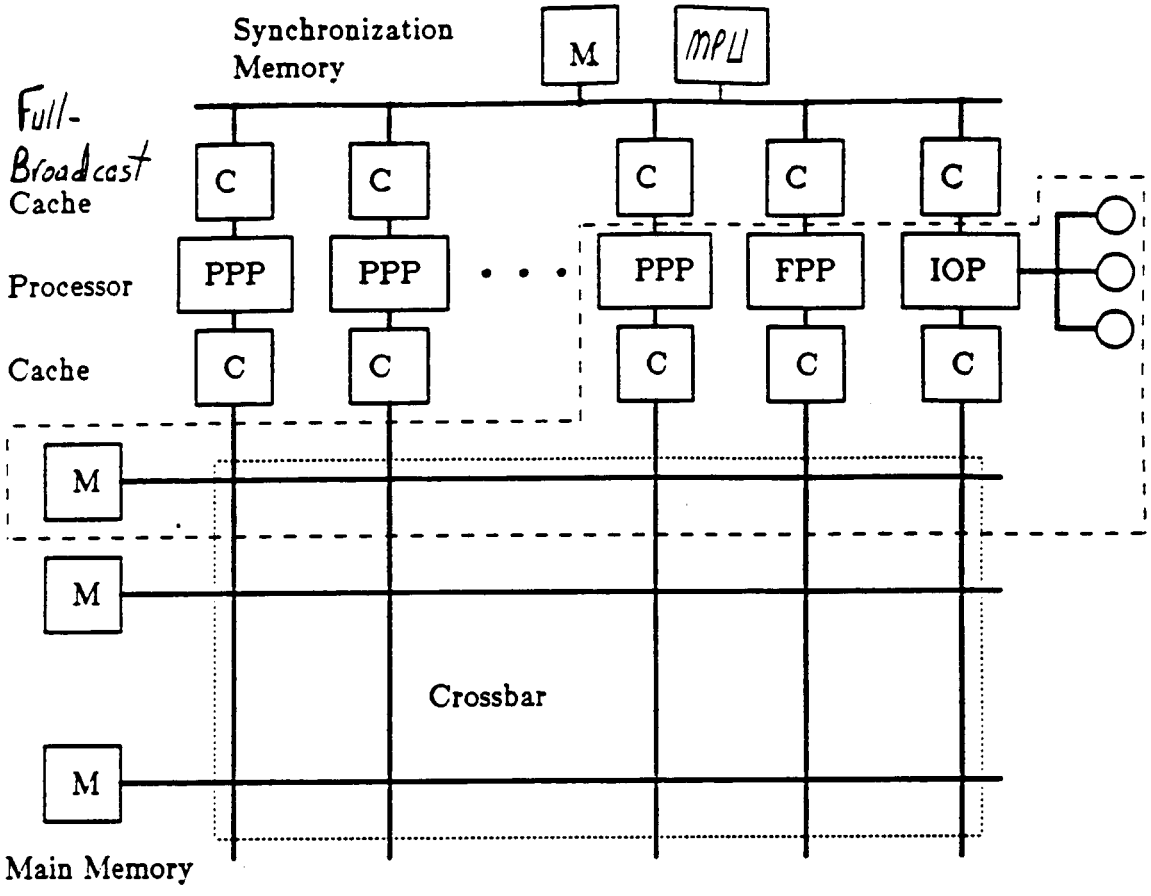



Figure 28. Aquarius Architecture.

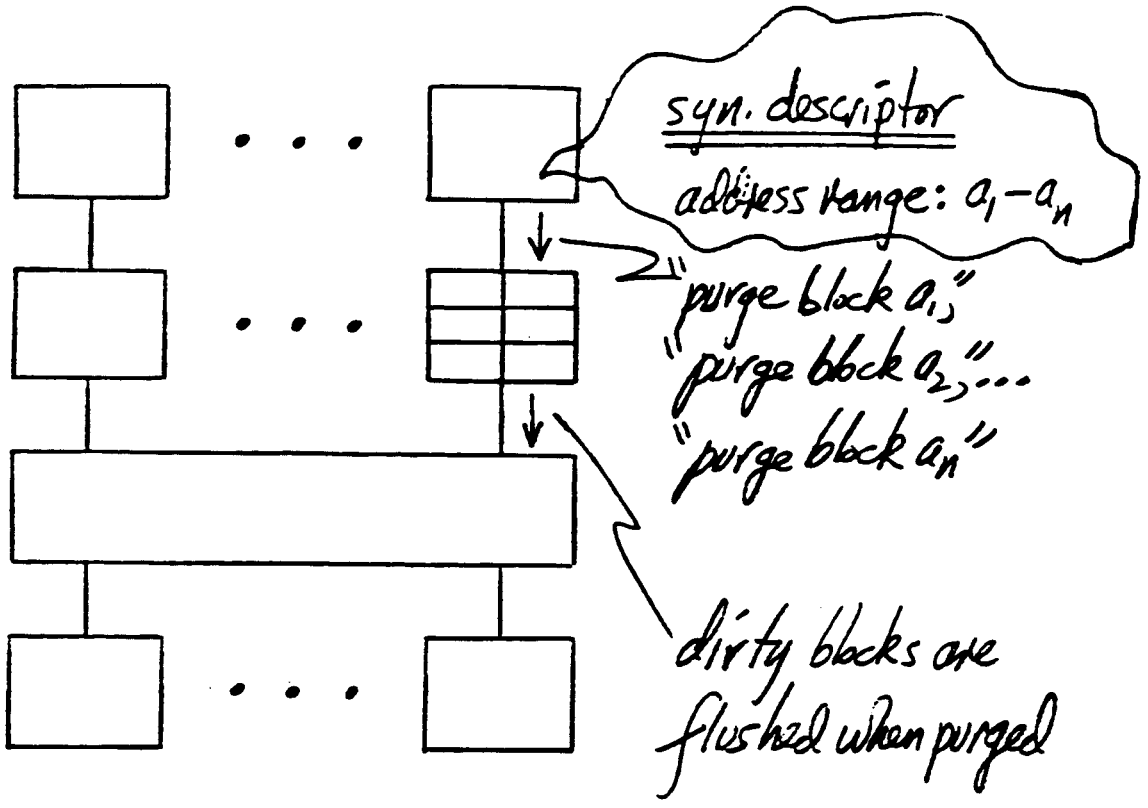



Figure 29. Providing latest version of soft atom: purge after use.

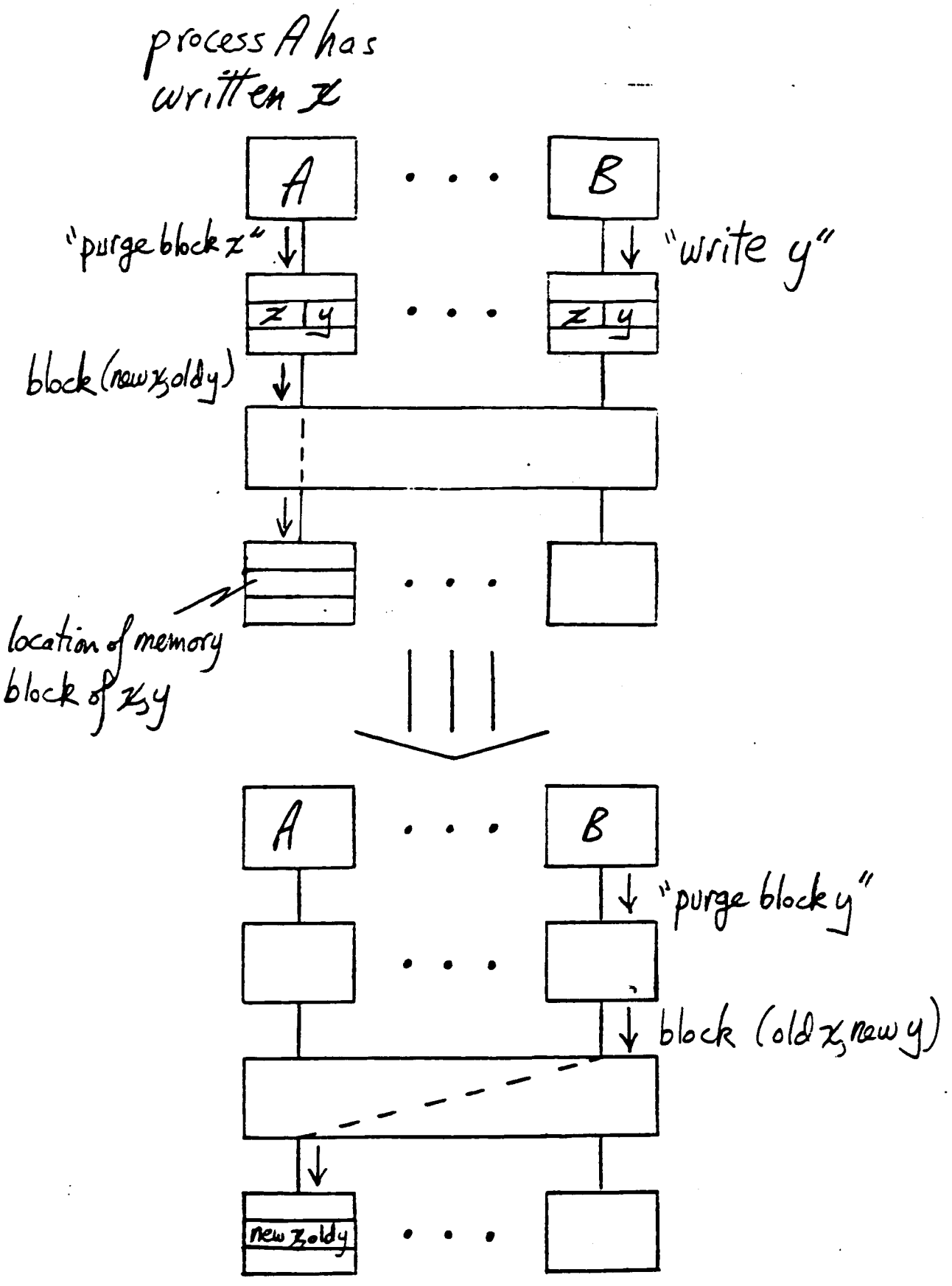



Figure 30. Error due to two cachable atoms on same block.